IN-61
193040
67 P

NASA Contractor Report 177625

Software Engineering Guidebook

John Connell and Greg Wenneson

CONTRACT NAS2-13210
September 1993

National Aeronautics and
Space Administration

NASA Contractor Report 177625

Software Engineering Guidebook

John Connell and Greg Wenneson

Sterling Software, Inc.
1121 San Antonio Road
Palo Alto, CA 94303

Prepared for
Ames Research Center
CONTRACT NAS2-13210
September 1993

NASA
National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

PAGE ______ INTENTIONALLY BLANK

TABLE of CONTENTS

TABLE of CONTENTS

TABLE of CONTENTS

LIST OF FIGURES AND TABLES

Section 1

INTRODUCTION

1.1 Identification

This is the Software Engineering Guidebook (SEG), a NASA Contractor's Report produced under the Computer Software Services (CSS) Contract, NAS2-13210 at NASA Ames. This Guidebook describes an orderly set of engineering methods for creating quality software on small to medium sized, non-critical risk projects under NASA funding.

1.2 Scope

Intended Audience

The guidebook is written for managers and engineers who manage, develop, enhance and/or maintain non-critical software in NASA environments.

Applicability

This guidebook is equally applicable to software development of new systems, enhancements to existing systems and maintenance of problems for an established software base. Although traditionally these areas of software engineering have been separated, for purposes of this guidebook, the separations are not maintained. All forms of software engineering share common elements of a repeated requirements definition, design, implementation, testing and release cycle. Development, enhancement and problem repair are essentially the same activity carried out at different times during the software's lifespan.

It is appropriate to reuse existing software during development of new applications. It is also appropriate to write new software to perform a requested software maintenance modification of an existing application. This blurring of the traditional distinction between development and maintenance names this the Software Engineering (not development) Guidebook.

Engineering Support Assumptions and Recommendations

This guidebook is written (and has an adopted a tone) on the assumption that software projects are supported by a separate (or possibly integral) Software Engineering Process Group (SEPG) responsible for providing and promoting improvements in the software engineering process. The SEPG is separate from Assurance and other quality improvement groups and oriented specifically to improving the means of engineering software. The establishment of a SEPG and its responsibilities are described in [FOWLER].

Site Customization of SEG methods

The guidebook provides a formalized Software Engineering Process with descriptions of common SEPG supported methods, activities, phases, and deliverables applicable to software engineering projects. The purpose of such formalization is to foster a duplicable process that can be applied consistently with predictably successful results.

This guidebook, as with any guidebook, should be customized and made appropriate for application to any specific site and environment. The supported methods described here can and should be supplemented or replaced entirely by a specific site's SEPG supported methods.

Supported vs. Approved Methods

Supported software engineering methods consist of one or a combination of the following three industry accepted methodologies: structured, object-oriented (OO) and rapid prototyping. Each of these three methods are defined by published tools, approaches, and definitions of products generated. The SEPG will support these methods by providing training and consulting. These methods are described in Section 4, "Supported Development Methodologies."

Software engineering methods not supported by the SEPG and SEG will be accepted for engineering software if they are supported in the data processing industry by published works and commercially available tools. Accepted methods may be used to engineer software but will not be supported by training or consulting provided by the SEPG.

Software Engineering Overview

A Software Engineering Process can be defined as prescribed *activities* conducted during defined life-cycle *phases* to produce specific products or *deliverables*. Software engineering activities are performed by software engineers using prescribed methods, techniques, and tools. Phases end when a review verifies that the activity for that phase is complete and the product is correct. Projects engage in activities in order to produce deliverables. Therefore, when a phase is over, associated deliverables must be complete and correct. Sections 3 and 4 provide explanations of the software engineering process and supported methods.

1.3 Purpose

This document provides clearly defined descriptions of methods for engineering software to meet the requirements of small to medium and non-critical risk tasks. These methods are still entirely appropriate for larger and/or higher and more critical-risk tasks, though the associated task structure, assurance and documentation requirements would be significantly larger and more complex.

1.4 Guidebook Status

This is version 2.0, dated May 31, 1993. This document was developed using the NASA Software Documentation Standard, Template NASA-DID-999. No sections are rolled out into subordinate volumes.

1.5 Organization

The guidebook's major sections are as follows:

Section 3, Software Engineering Life Cycles and Methodologies, contains an overview of approaches and techniques for developing quality software.
Section 4, Supported Development Methodologies, describes specific SEPG supported development methodologies.
Section 5, Assurance, describes standards, reviews and testing techniques for ensuring development of quality software.
Section 6, Configuration Management, describes configuration control and change management approaches to ensure known and controlled configurations and releases.

Section 2

RELATED DOCUMENTATION

Applicable Documents

[AHB] Ames Handbook 5333.1, Establishment of Software Assurance Programs, May 1992, NASA Ames.

[DOCSTD] NASA Software Documentation Standard, NASA-STD-2100-91, July 1991, Office of Safety and Mission Quality.

[NMI] NASA Management Instruction 2410.10, NASA Software Management, Assurance and Engineering Policy, March 26, 1991, NASA Office of Safety and Mission Quality, Washington, DC.

[SAG] Software Assurance Guidebook SMAP-GB-A201, September 1989, NASA Office of Safety, Reliability, Maintainability and Quality Assurance, Washington, DC.

[SFIG] Software Formal Inspections Guidebook (Draft), June 1990, NASA Office of Safety and Mission Quality, Washington, DC.

[SFRG] Software Formal Reviews Guidebook (Draft), June 1990, NASA Office of Safety and Mission Quality, Washington, DC.

[SIPSTD] NASA Software Inspection Process Standard, NASA DS-xxxx-91 (Draft), March 24, 1992, Office of Safety and Mission Quality.

[SMPG] Software Management Plan Guidebook (Draft) February 1992, System Technology Institute developed for NASA/Lewis.

[SQAG] Software Quality Assurance Audits Guidebook (Draft) June 1990, NASA Office of Safety, Reliability and Quality Assurance, Washington, DC.

[SWASTD] NASA Software Engineering and Quality Assurance Standard, NASA DS-xxxx-91 (Draft), December 1991, Office of Safety, Reliability and Mission Quality.

Information Documents

[BABICH] Babich, W., Software Configuration Management: Coordination for Team Productivity, Addison Wesley, 1986.

[BASILY] Basili, V. & Selby, R., Comparing the Effectiveness of Software Testing Strategies, IEEE Transactions on Software Engineering, Vol SE-13 No. 12, December 1987.

[BEIZ84] Beizer, B., System Testing and Quality Assurance, New York: Van Nostrand Reinhold, 1984.

[BEIZ83] Beizer, B., Software Testing Techniques, New York: Van Nostrand Reinhold, 1983.

[BOEHM] Boehm, B., Software Engineering Economics, New Jersey, Prentice-Hall, 1981.

[BROOKS] Brooks Jr., F., The Mythical Man Month, New York: Addison Wesley, 1975.

[COAD-OOA] Coad, P. & Yourdon, E., Object-Oriented Analysis, New York: Yourdon Press (Prentice-Hall), 1990, 1991.

[COAD-OOD] Coad, P. & Yourdon, E., Object-Oriented Design, New York: Yourdon Press (Prentice-Hall), 1991.

[CONN89] Connell, J. & Shafer, L., Structured Rapid Prototyping, Englewood Cliffs, New Jersey: Yourdon Press (Prentice–Hall), 1989.

[CONN91] Connell, J., Gursky, D. & Shafer, L., Object-Oriented Rapid Prototyping, 2 parts, Embedded Systems Programming, September and October, 1991.

[CONN92] Connell, J. & Eller, N., Object-oriented Productivity Metrics, Proceedings of the Ninth Annual NASA Contractor Conference on Quality and Productivity, October 1992.

[CONST] Constantine, L. & Yourdon, E., Structured Design, New York: Yourdon Press (Prentice-Hall), 1978.

[DeMARCO] DeMarco, T., Controlling Software Projects, New York: Yourdon Press (Prentice–Hall), 1982.

[DREGER] Dreger, J., Function Point Analysis, New Jersey: Prentice Hall, 1989.

[FREED] Freedman, D. & Weinberg, G., Handbook of Walkthroughs, Inspections and Technical Reviews, New York, Dorset House, 1990.

[GRADY] Grady, R. & Caswell, D., Software Metrics, New Jersey: Prentice Hall, 1987.

[HARROLD] Harrold, M. & McGregor, J., Incremental Testing of Object-Oriented Class Structures, Proceedings of the 14th International Conference on Software Engineering, 1992.

[JONES] Applied Software Measurement, Jones, C., New York: McGraw Hill, 1991.

[McMENN] McMenamin, S. & Palmer, J., Essential Systems Analysis, New York: Yourdon Press (Prentice-Hall), 1984.

[MEYER] Meyers, G., Software Reliability, New York, Wiley & Sons, 1976.

[SCHLAER] Schlaer, S. & Mellor, S., Object-Oriented Systems Analysis, New Jersey Prentice-Hall, 1988.

[FOWLER] Fowler, P. & Rifkin, S., Software Engineering Process Group Guide, CMU/SEI-90-TR-24, September 1990, Software Engineering Institute.

[SEI-92] Guidance on Understanding and Tailoring the NASA Software Documentation Standard (Draft), February 1992, System Technology Institute developed for NASA/Lewis.

[SMAP4.3] Information System Life-Cycle and Documentation Standards, Release 4.3 February 28, 1989, Software Management and Assurance Program, Office, NASA Office of Safety, Reliability, Maintainability and Quality Assurance, Washington, DC.

[SMITH] Smith, M. & Robson, D., A Framework for Testing Object-Oriented Programs, Journal of Object-Oriented Programming (JOOP), June 1992.

[WARD] Ward, P. & Mellor, S., Structured Development of Real-Time Systems, New York: Yourdon Press (Prentice-Hall), 1985.

[YOUR84] Yourdon, E., Modern Structured Analysis, New York: Yourdon Press (Prentice-Hall), 1984.

[YOUR83] Yourdon, E., Structured Walkthroughs, New York: Yourdon Press (Prentice-Hall), 1983.

Section 3

SOFTWARE ENGINEERING LIFE CYCLES and METHODOLOGIES

3.1 Software Engineering Approach

Software Engineering is the creation of software products by applying systematic engineering techniques, assurance activities and configuration control methods over the life cycle of a product from conception through development, operation and maintenance to eventual retirement. Application of general engineering techniques entails dividing a complex problem into achievable and manageable pieces, and for each piece, applying specific methods to generate known and verifiable products during each phase.

Correct application of software engineering activities should produce software that performs the intended job properly. The software engineering process is meant to produce a quality product that is free from errors. The entire process should be controllable, repeatable and predictable.

The following subsections will provide a summary of the information contained in the remainder of the guidebook. The principles in each section must be addressed for any software engineering task. The selection of the methods and approach appropriate for a particular project is a combined engineering and management responsibility. These approaches and methods should be detailed in each project's Management Plan.

3.2 Software Engineering Life Cycles

All development activities are part of a process (a series of actions, changes or procedures) that generates products or results. If the process is chaotic (disorganized or confused), this may accidentally lead to a successful conclusion. If it is coherent and follows a comprehensible sequence, it tends to lead more often to a predictably successful result.

Figure 3.2-1 is the graphical depiction of a Software Engineering Process. Such a depiction is often referred to as a life cycle because it shows all the activities that can occur from "cradle to grave" within the life span of a piece of software. Each phase shows the SEPG supported software engineering methods and how testing activities fit into the life cycle.

Depending on the size, complexity, risk and development methods selected, the phases of a development process can be structured with more or less complexity than shown. A large complex system product with several subsystems may have the Software Detailed Design Phase broken into preliminary and detailed design and implementation phases, one set for each subsystem. Additionally the System Integration phase might consist of several subsystem integrations leading to a full system Test and Integration phase. A small development may only consist of two discrete activities before delivery: requirements definition and design/implementation (including testing).

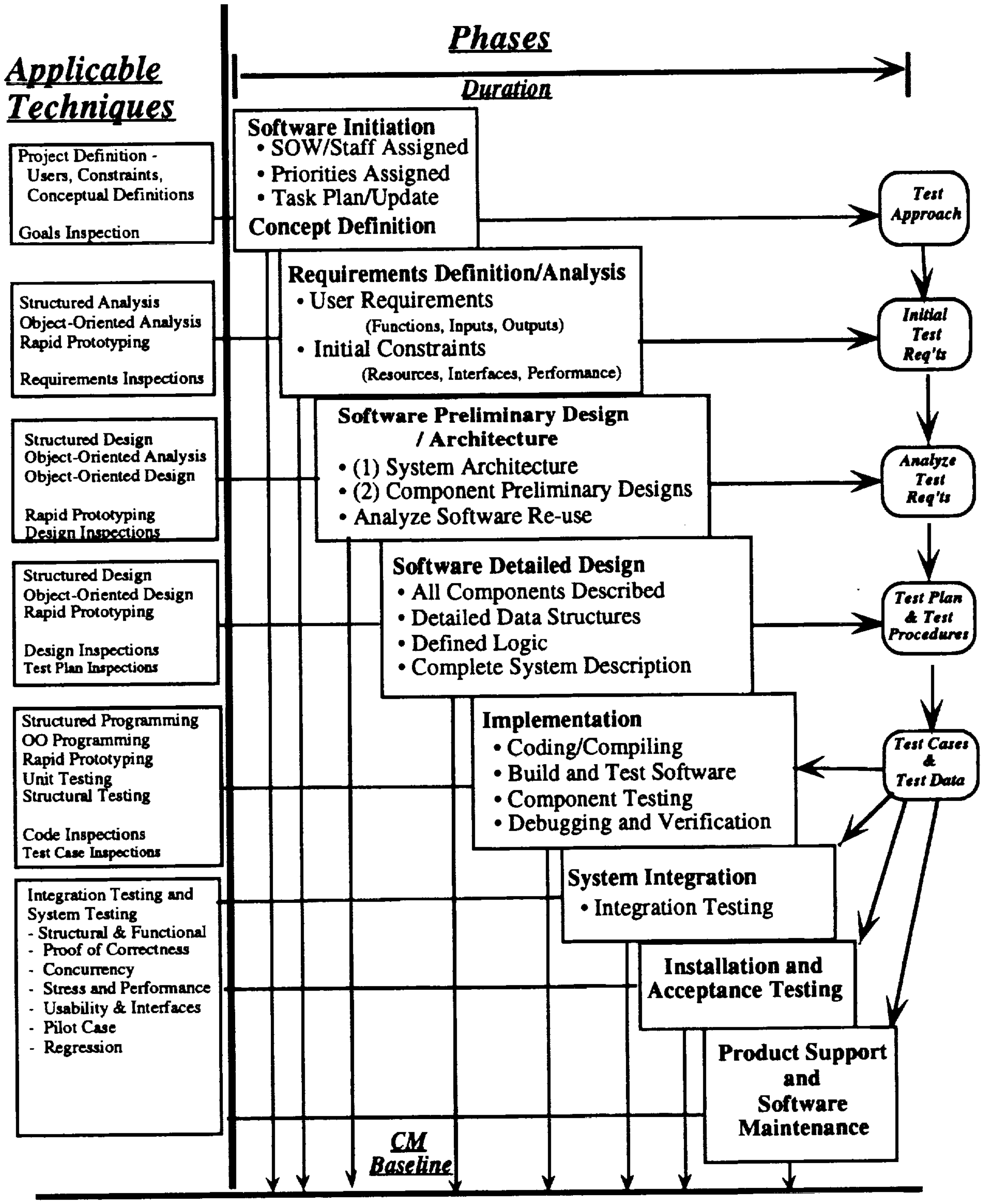


Figure 3.2-1 Software Engineering Process Model

3.3 Software Engineering Methods

The methods described in this guidebook are supported by the SEPG; that is, the software engineers will provide or arrange training and consulting for these methods. The supported methods were selected because they are commonly available and accepted engineering techniques which have been published and are supported by commercially available tools. Use of the supported methods do not guarantee a successful project but do increase the capability to determine the customer's true requirements and provide an appropriate product that meets customer satisfaction.

Other software methods may be used to engineer software. Though these methods may not be supported by the SEPG training and consulting staff, they can be accepted for use if they are used in the industry, published and available, and have commercially available support tools.

3.3.1 Supported Methods

Each SEPG supported engineering activity is to be completed using one or a combination of three basic approaches: structured, object-oriented (OO), or rapid prototyping (RP). These approaches may be used exclusively and consistently; e.g., 100% structured development. Alternatively, techniques may be combined for a particular project to suit the needs of a particular application or development environment. For instance Object-Oriented Analysis (OOA) and Design (OOD) can be effectively combined with structured programming. Structured or Object-Oriented techniques can be combined with rapid prototyping and do structured rapid prototyping or object-oriented rapid prototyping. Any such reasonable combination will be supported by the SEPG.

Each of these three basic techniques is defined by the tools and the products of those tools when used by developers in application development. In other words, you can tell if an engineer is doing Structured Analysis (SA) or Object-Oriented Analysis by looking at the tools they are applying to the Requirements Analysis activity and whether Data Flow Diagrams or Object Information Models are produced. The tools and products applicable to Structured and Object-Oriented techniques, respectively, are shown in Section 4, "Supported Development Methodologies".

3.3.2 Concurrent Engineering

In Figure 3.2-1, the engineering phases overlap. In the older waterfall development life cycle models, each of the phases were supposed to complete with a formal review before any work on the next phase began. Contemporary engineering approaches generally overlap the phases in a concurrent rather than sequential manner. The actual amount of overlap can vary from almost none to almost 100%. Defining known configurations (baselining) and controlling them (configuration management) are still required before proceeding, as shown in Figure 3.2-1. Concurrent engineering has several benefits which can lead to higher quality products:

- Rework is reduced because all skills are available on the task and communication about implementation feasibility can occur more readily;
- Consistency with most published software engineering standards, including NASA and DoD, which now encourage concurrent engineering during software development;
- With one underlying graphic representation, Object-oriented Analysis and Object-Oriented Design are really one activity with concurrent prototyping recommended;
- If you prototype, you are already doing it: you must have a design to produce a prototype, yet prototyping is a requirements discovery technique.

3.4 Documentation

Good documentation is a software engineering tool. Documentation is part of the project work, part of the project deliverables, and part of the tools that produce quality products. It is not an extra burden added onto coding or design phases of development. Good documentation accomplishes several purposes during the life cycle:

- Can reduce development costs by providing correct, complete, and exact blueprints for programmers, which allows implementation and testing to take less time while producing better results.
- Can reduce subsequent maintenance and enhancement costs by providing correct, complete, and exact models for programmers to use to understand, design, and test, which allows implementation and testing to take less time while producing better results.
- Acts as a tangible demonstration that early design work has been accomplished.
- Provides a stand-alone representation of the work, which can be reviewed or inspected independent of the author's presentation.
- Provides additional understanding of the product or its operation not obvious in the product itself.
- Provides protection from the "big bus" threat (if the developer steps in front of a big bus, will any tangible work remain?) and allows staffing changes with minimal impact.

Documentation must be **useful and appropriate for the product and project** and satisfy any standards or policies such as AHB 5333.1. Documentation blindly produced seldom serves anyone and may be a complete waste of time. Appropriate documentation is easy to read and communicates significant ideas to those who have a need. Document organization should be layered; with larger scope concepts presented first and more detail available later. Managers needing an overview should not need to read as far as a programmer.

A preliminary User Guide should be produced early in the development. The User Guide can act as a prototyping tool to help formulate the initial system concepts and expectations of the system's appearance and capabilities prior to requirements definition. The user guide only deals with the interfaces: inputs, outputs, and visible response of the system.

All projects' documents should follow the organization, format, and content placement described in the NASA Software Documentation Standard templates, called "DIDs" (Data Item Descriptions), which are specific for particular documents (e.g. NASA-DID-P200 is the format for a requirements document, NASA-DID-P600 is for a User's Guide). The DIDs do not require that a specific methodology be used. DIDS only provide a standard format and organization for the developed information.

Different development methods will populate the documents with different information or provide a different emphasis. Even though the DIDs provide the general format and content required, an object-oriented development will produce a Product Specification with different content than that of a structured development because the tools and methods are different. The SEPG has specific examples for using the DIDs in different development methods.

Automated documentation support should be considered an essential part of the development environment. Document production should be a byproduct of the requirements, design, and implementation processes, and well-supported by the tools that help generate the software designs and products. Documentation *information* are the requirements, design and the comments in the code. If documentation is a required deliverable, it should be assembled from the requirements, design and code.

3.5 Assurance

A quality product is measured by how well it meets customer expectations. Using methodical development processes will help properly determine customer expectations and document them as requirements, and generate a design and product which matches those expectations. Assurance activities performed in parallel with and as part of the development will set and check conformance to standards to ensure that the development and its products are proceeding as planned.

Assurance activities are performed on products over the entire development life cycle. The primary assurance activities covered in this guidebook are: Risk Analysis, Quality Assurance (QA), Verification and Validation (V&V) and Problem Reporting. All of these assurance activities should be addressed by any task developing or maintaining software.

Risk Analysis must be performed as part of good engineering practices and in response to NASA, see [NMI] and Ames requirements, see [AHB]. Specific risks must be identified, evaluated, and mitigation planned. The degree of risk inherent in the development or in the end use of a product will determine the rigorousness of development and assurance. Higher risk projects and products require more care and checking than low risk projects and products. The identified project and product risks and the planned measures to reduce risk must be documented in the project's Management Plan.

Quality Assurance (QA) is the setting of product and process standards of performance and the checking for conformance to those standards. Each project should establish development standards appropriate for the environments. Baseline (or beginning) standards can be obtained from a variety of sources (Company standards, bookstores, etc.) and should be specifically adapted to the projects environment and goals. AHB 5333.1 establishes minimum standards for risk determination, documentation and reviews.

This guidebook provides guidelines for SEPG supported development methods. A project manager should select a methodology, customize it as necessary, and make it standard for the project. The standard should be checked as part of all technical reviews.

Verification & Validation (V&V) are the means to ensure that a product which does the right job is produced. V&V is usually done by design reviews, simulation, and testing. Reviews are required by Ames Assurance Requirements [AHB]; the types of reviews are determined by project risk level. Software Formal Inspections can be a very effective technical review process for finding errors. Testing and test planning should be an integral part of the development, but should not be the primary error detection mechanism. Both reviewing and testing assume an orderly and known development process is being followed.

Problem reporting and resolution should be standard procedure for every task that releases and/or supports products. Problem reporting is especially critical for products in pre-release testing and for the first year after release. Error discovery rates over time provide valuable indicators of the quality of the product and development process.

3.6 Configuration Management

Configuration Management (CM) provides the activities which are key to orderly release, testing, problem fixing and maintenance. The primary components of CM are identification and control of software products (and collections of products) and the control of changes to those products. Given this CM information and adequate back ups, any defined product version can be returned to or compared against another version.

CM is part of development activities and may be performed by the developers or a non-development group. CM provides an orderly development environment, assistance in problem resolution during development or testing, and provides planned, duplicable releases from known components. Poor CM can contribute to wasted time and effort by ensuring lost components, redundant or lost changes, difficult integrations, unrepeatable releases of unknown quality, and unacceptable time spent fighting the development environment and/or the overly strict CM environment.

3.7 Software Reuse

Software Reuse provides an excellent opportunity for improved productivity through higher quality or lower cost. Reuse of known and tested components can increase reliability and decrease development time. Reuse can be at the code, design, or concept level. COSMIC, Ames NASABBS and Ames SOFTLIB are libraries of known components available for reuse. The components come from a variety of applications and environments. Several larger projects such as the Ames Standardized Wind Tunnel System (SWTS) and the Aerodynamics Division Research Support groups have developed their own libraries of commonly used utilities and functions.

The greatest possibilities for reuse of components are within projects or applications of similar types (similar problem domains). Each development project should become familiar with similar projects' applications and will search the reuse libraries for reusable components as part of development activities. Preparation for and conduct of reviews and inspections should include the questions: "Can this component be reused or can this design reuse an existing component?" In addition, each project should develop a common library of utility functions which can be reused within the project. Reusable components will also be submitted to external reuse libraries.

Reuse needs to be planned for and actively managed; it does not happen by accident. There are very few totally new applications. With care, we can build from and upon already proven components. While object-oriented development provides a means to better define components and make them more easily reusable by providing better packaging, each task currently has the technology for effective reuse. However, effective reuse must be an active part of development plans and designs.

3.8 Software Metrics

It is very difficult to make a vigorous, plausible and job risking defense of an estimate that is derived by no quantitative method, supported by little data and certified chiefly by the hunches of the managers.
Fred Brooks, The Mythical Man Month

To estimate with a ±10% accuracy, historical data must be accurate to ±5%
Capers Jones, Applied Software Measurement

Metrics are measurements and numbers collected over time, which can provide valuable indicators of product and development process quality. One of the precepts of Total Quality Management (TQM) and Productivity Improvement is that you cannot manage what you can't measure. With the increased emphasis on improving productivity and quality, measurements of development processes, productivity and the quality of the products generated are of increasing importance.

Any effort or program to gather project or product **measures must have a real-world useful purpose**. Numbers gathered indiscriminately for reporting purposes only and without a real-world use basis can be counterproductive. Any metrics gathering must have staff understanding and be used in a non-threatening manner to receive their support. Although many numbers and measures can be gathered, they are just data unless they are put in a meaningful context; then the data becomes information.

Many project measures are already available from orderly and planned developments. Some of these are: initial schedule and work estimates, time spent in various phases, software inspection and review statistics, product size, and problems reported. More sophisticated use of some of these numbers include: requirements change frequency to indicate requirements definition effectiveness, problem detection rate to show the effectiveness of testing or readiness for release, and pre-and post-release problems and problem detection rate. Grady and Casswell in Software Metrics [GRADY] provide good information on establishing a realistic metrics program for a project.

Every development project should collect and maintain a record of as many of the following measures as practical within staffing and budget considerations:

- Estimates of staffing, development phase schedules, and labor hours required.
- Actual calendar start, time and completion dates for phases, when staff added, Engineering time spent by activity or development phase where time spent and how: meetings, training, design, code, fix, test. If possible, a further breakdown of time spent allocated to subsystem or lower level module.
- Some measure of the changes introduced from external or from internal sources and the reason for the changes.
- Status of Code Modules/Product: inspections held and type, current development status, design/code reuse.
- Size of and numbers of software units, subsystems, and systems by a quantitative metric. This should include all delivered and even non-delivered code or product.
- Defect tracking information for pre- and post-release problems: type of error, reason, specific unit, severity (see Section 5.6, "Problem Reporting"). Individual developers should be encouraged to keep informal defect metrics for their developed modules prior to integration where project defect tracking would start.

These basic measures of how much effort is being expended and where, will focus attention on effective managing, product quality, and productivity of the development process. When examined over time, this information can provide pointers to effectiveness of estimating, trend information, and error rates. By identifying early problem areas through review and inspection statistics, continuing problem areas in implementation and testing can be predicted.

With these measures, comparison can be made to existing published metrics such as SPR's Capers Jones' Productivity metrics (see section 3.8.2 below). These comparisons can provide guidance to improving the product and development process.

The measures can be informal and not require great effort or large support systems to collect and retain. They need not even be revealed to higher management. If for nothing else than to apply the Hawthorne effect (positive attention to staff improves attitude which improves productivity), these metrics should make development more effective and generate higher quality products than if these areas are ignored.

3.8.1 Quantitative Metrics for Software

To provide the basis for accurate quality and productivity comparisons, metrics must provide nonambiguous and consistent measures across different development methodologies, languages and implementation environments. Halstead, McCabe, and Function Point metrics can provide some unambiguous indicators of software program complexity and therefore (from history) a prediction of problem areas. Lines of code (LOC) can be used as a basis for measuring software if no other supporting measures are available.

LOC are generally not a good indicator of software attributes. However, they are simple to use and can be helpful for measuring error rates or other simple ratios within a project. They should be applied consistently within specific areas of a project. They are not dependable between or outside projects unless the same standard of use is defined and adhered to. Automated LOC and comment counting utilities are commonly available. Counting LOC is counterproductive for measuring productivity, especially where reuse or high level development environments are involved.

As programs become more complex they have proven to be more error prone and have higher maintenance costs. Halstead and McCabe metrics provide an indication of module and program complexity and thus a pointer to future error prone areas.

Halstead software science metrics (1977) give an indication of program complexity and length based on the richness of vocabulary. A complexity figure for a section of code is obtained from the counts of different operands and operators and their frequency of use. Higher counts are more complex. An analogy is that scientific papers are more complex than the daily newspaper because the language used is richer in complex terms.

McCabe cyclomatic and essential complexity methods (1976) and numerous refinements provide a measure of program inter- and intramodule complexity based on the number of logical paths available. The more logical paths and branching possible, the greater the complexity. It does not take into account processing or data structure complexity.

Halstead and McCabe metrics are usually supported within automated test coverage tools (tells how much of a program's code was exercised by test cases). The metrics point out complex sections of code which may be need to be re-engineered or scheduled for more comprehensive testing. The metrics are not laws, but indications of areas of potential problems.

Albrecht's Function Points (1979, revised 1984) and many following refinements are based on the externally visible aspects of a program such as inputs, outputs, inquiries, logical files, and interfaces. Function Points (FP) provide a good, language-independent means for measuring software size. A defined functional capability will have the same FP count whether implemented in Ada, assembly or C++. FPs provide a standard, reliable and unambiguous means to measure, compare, and evaluate process and product quality and productivity. Dreger's *Function Point Analysis* [DREGER] provides a good description of how to use FPs.

Feature Points extends Albrecht's Function Points to make it applicable to real-time, embedded, and systems software. The Feature Point extension adds an adjustment factor for complexity in the problem, code, and data. There are a number of other variations on FPs to allow for different requirements or complexity scenarios; all seem to come within a 20% variation range.

Object-Oriented Effort Points (OOEPs) were proposed in 1992 by John Connell and Nancy Eller [CONN92] of Sterling Software to bring unambiguous measurement to object-oriented analysis and design. OOEPs are a unit of measure of complexity and development effort required for object classes. OOEPs are determined by assigning values and counting based on the class attributes, the services or methods provided and how many external entities the object will deliver data to or get data from. Additionally, more OOEPs are assigned if the object is computationally intensive. OOEPs can be determined during the object-oriented analysis or design phases and, similar to FPs, can predict how much implementation effort is required and where special review and test attention is needed.

As of this writing, none of the Function, Feature or Object-Oriented Effort Points are supported by automated function point counting from code or design, although one commercial vendor of metrics methodologiers has indicated intent to provide FP counting support from Data Flow Diagrams.

Quantitative measures for software size and complexity provide immediate benefits in analysis and design phases with:

- a capability of predicting where the errors will be;

- a prediction of actual code size;
- an indication of where more review and test attention is needed and the number of test cases necessary to provide logical coverage;
- indications of whether design units are too complex for their function and should be redesigned to reduce unnecessary complexity.

3.8.2 Productivity Metrics

Although there are no productivity metrics for Ames, Capers Jones, of Software Productivity Research, Inc. (SPR), has collected world-wide metric data on over 4000 software projects over several decades. SPR consults, teaches, and publishes information on productivity and measurement. Part of SPR's consulting is a thorough examination and evaluation on company management and development practices, so that metrics information is both extensive and in-depth. SPR's metrics are probably the most comprehensive of any on which information is published. Information published from the data includes a full range of productivity metrics and the factors which affect productivity and quality. Much of this information is available in Jones' 1992 book, *Applied Software Measurement* [JONES].

Before any meaningful productivity comparisons can be made, normalized values for the compared items must be available and historically validated (for the environment). This is why measurement programs require a time history to be effective and are not up and running overnight. Measures of quality (such as problems per functional unit of code) also require a time history and experience to be useful. Capers Jones estimates from experience that most measurement data is incomplete and totally misses 30 to 75% of actual cost and effort. Basing any measurement comparison on industry or internal numbers requires that both sets of numbers be understood and comparable.

Section 4

SUPPORTED DEVELOPMENT METHODOLOGIES

This section describes the overall software engineering approach to be used on the CSS contract. It describes, methods and techniques that are currently approved and supported by the SEPG and that you are encouraged to employ during software development. A single methodology is not mandated. The SEPG will support the use of structured, object-oriented, or rapid prototyping approaches as defined in this section. Developers are free to mix and match components from the following paragraphs in any way deemed sensible for their tasks. The supported methods are not limited, in terms of usefulness, to new development or production (non-research) software. Enhancements to or major adaptations of existing software can also benefit from use of these techniques. Artificial Intelligence (AI) research support software using *fuzzy* logic can benefit from use of these techniques. Using these techniques should enhance productivity, reducing rather than increasing staffing requirements for any size project. Developers are encouraged to contact an SEPG representative for consultation as to what is sensible for their task.

4.1 Structured Development

If the software developer chooses to apply structured techniques to the software development process, then the appropriate mapping of techniques to phases in the Development Process Model in Figure 4.1-1 is as follows:

Phase	**Technique**
Requirements Definition/Analysis	Structured Analysis & Inspections
Software Preliminary Design	Structured Design & Inspections
Software Detailed Design	Structured Design, Structured Testing & Inspections
Implementation	Structured Programming, Structured Testing & Inspections
System Integration	Structured Testing
Installation & Acceptance Testing	Structured Testing

These techniques are described in sections 4.1.1 through 4.1.3.

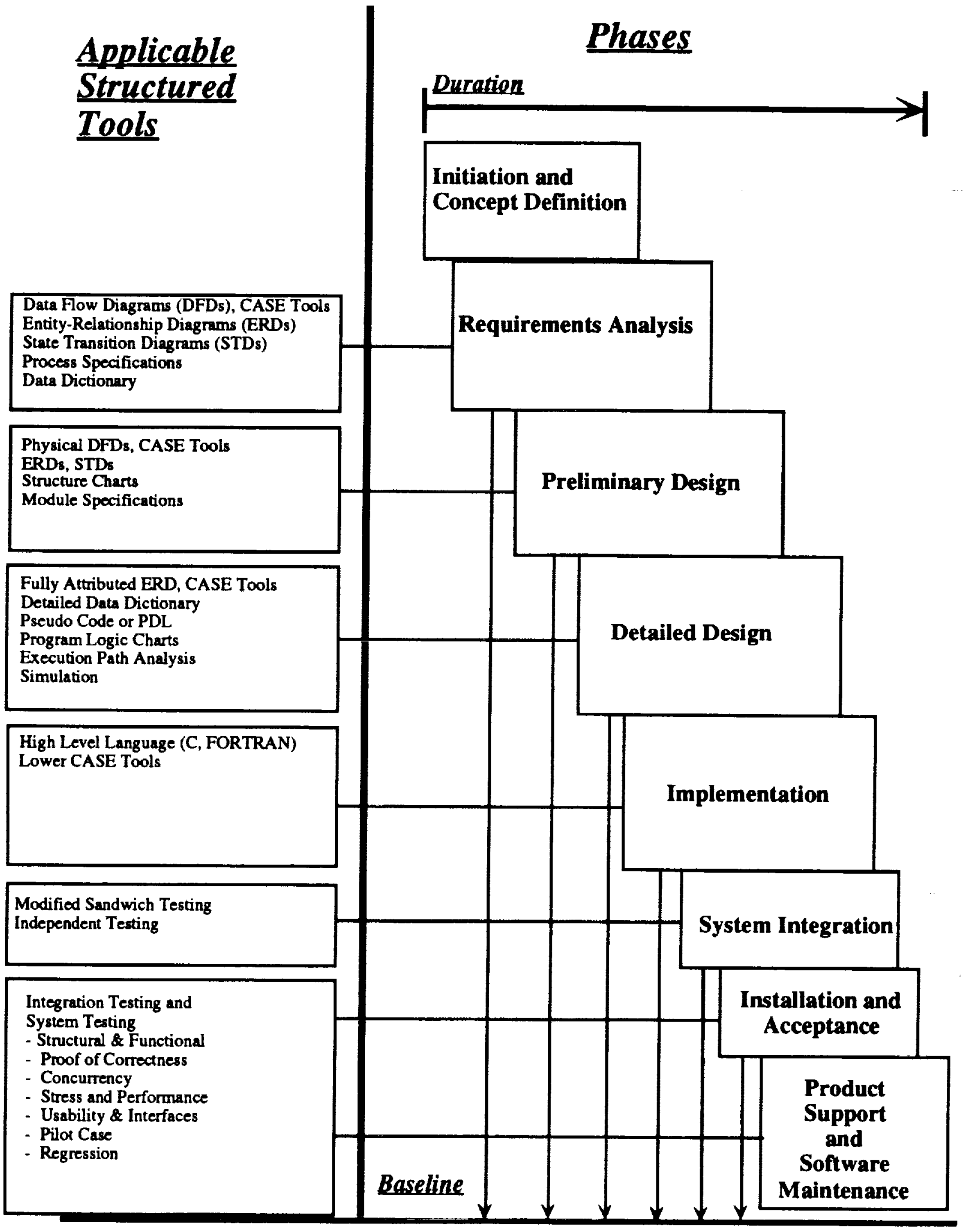


Figure 4.1-1 **Structured Development Process Phases and Tools**

4.1.1 Structured Requirements Analysis

The following paragraphs briefly describe tools that can be used to prepare a structured requirements specification of the software's proposed functionality. The SEPG supports an approach to Structured Analysis usually referred to as Yourdon/DeMarco as well as an approach to Real-Time Structured Analysis referred to as Ward/Mellor. These are probably the most widely used Structured Analysis approaches, but it is important to note that acceptable variants also exist, such as Gane and Sarcen, IDEF, SADT, and Hatley-Pirbhai. Several fairly current texts are excellent references for learning and applying this approach, in particular see [DeMARCO], [McMENN], [WARD], and [YOUR84].

4.1.1.1 Dataflow Diagrams

Dataflow diagrams (DFDs) are a central part of most structured analysis approaches. They represent a hierarchical decomposition of application system functionality with depictions of data interfaces between the functional components. Diagramming begins with a context diagram and proceeds to decompose the graphically represented processes in descending level diagrams from level zero to level n, where n is a number that varies according to the size of the application being modeled. Level "n" is referred to as the primitive level; a guideline for determining that the primitive level has been achieved is: if the specifications for all processes on this level can be written using at least half a page, but no more than one full page of structured English. More decomposition than this will cause the model to become overly complex. Less decomposition will cause individual specifications to become overly complex. The data interfaces, called dataflows, are all defined in a data dictionary using a non-ambiguous data dictionary syntax. Dataflow diagrams contain graphic symbols representing:

- external information sources and destinations (rectangles on context diagram only)
- software processes (circles)
- data flowing from source to destination (directed vectors between externals, processes, and data stores)
- data stores (sets of parallel lines)

Figure 4.1-2 is an example of a context diagram and Figure 4.1-3 is an example of a lower level decomposition of that diagram.

4.1.1.2 Entity-Relationship Diagrams

Entity-relationship diagrams (ERDs) are a part of some structured analysis approaches, for example, real-time structured analysis (RTSA). They are used to represent the internal data structure of a complex data store or a set of related data stores. There is not a hierarchy of diagrams as with dataflow diagrams, but rather single-level diagrams representing a data store as a collection of data entities and their required relationships. Entity-relationship diagrams contain graphic symbols representing:

- data entities (rectangles)
- relationships (lines connecting entities)
- cardinality (notations such as 1,m at the end of a line)

Figure 4.1-4 is an example of an entity-relationship diagram depicting the structure of the data store shown in Figure 4.1-3.

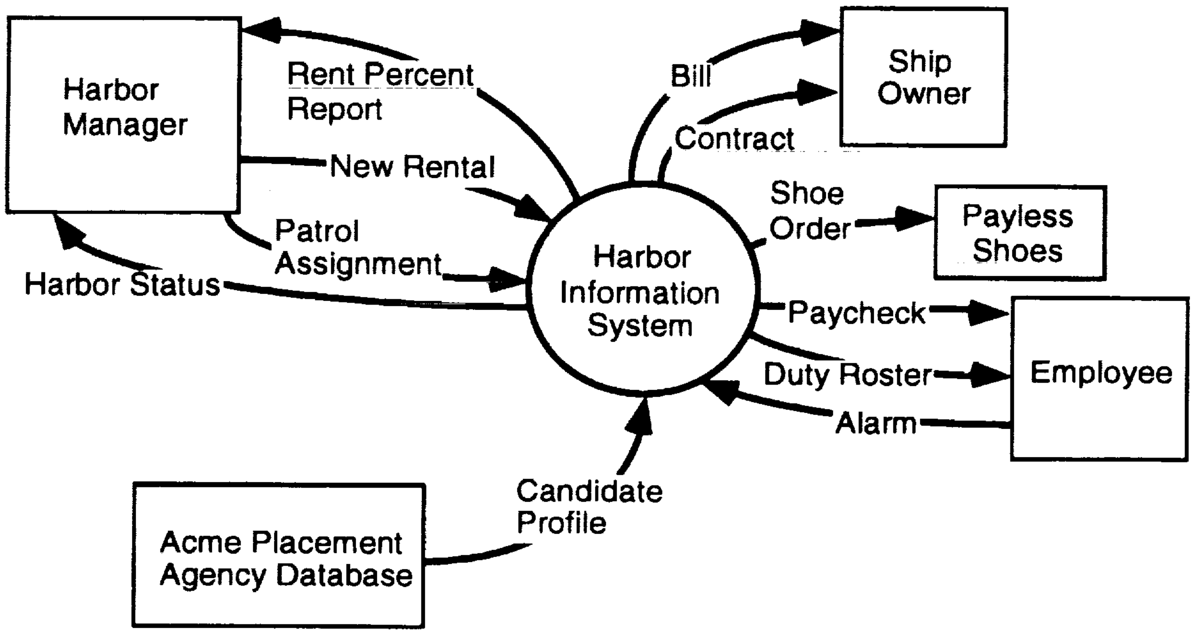


Figure 4.1-2 **Context Dataflow Diagram Example**

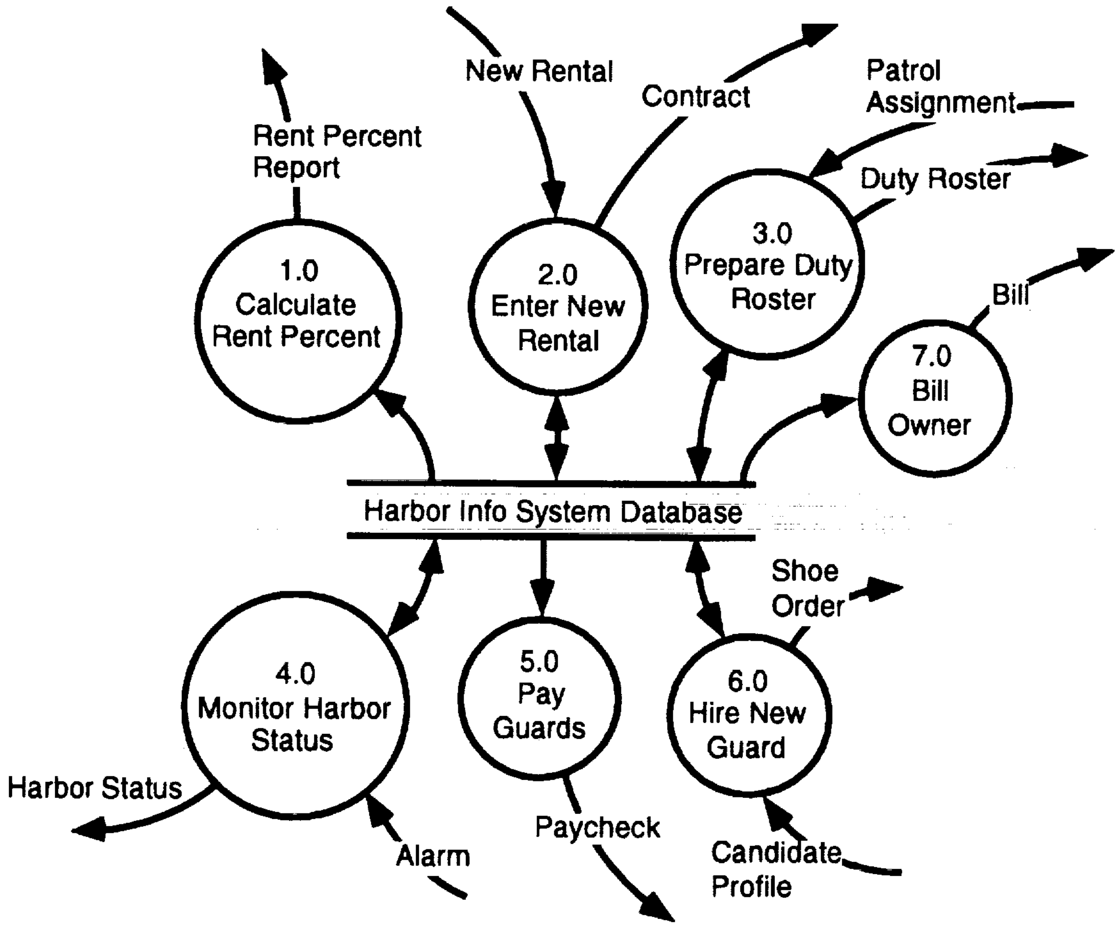


Figure 4.1-3 **Example of Functional Decomposition Using a Dataflow Diagram**

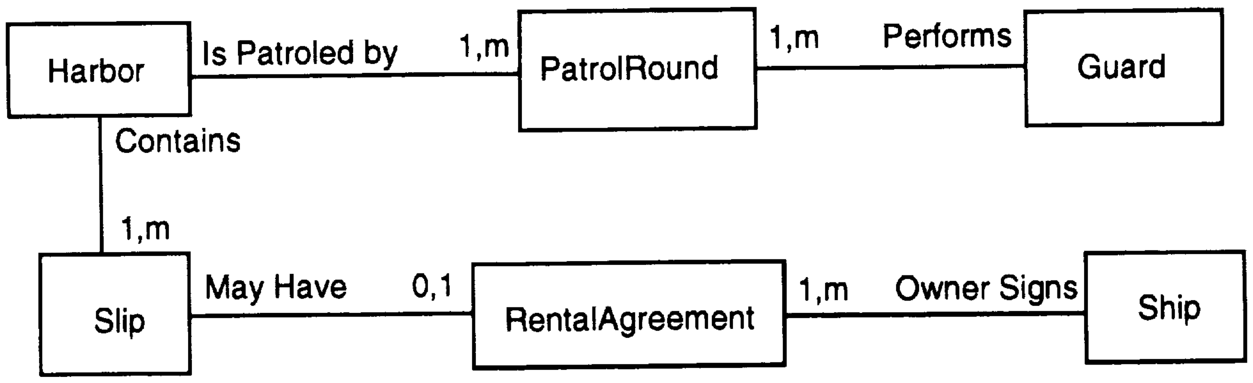


Figure 4.1-4 Entity-Relationship Diagram Example

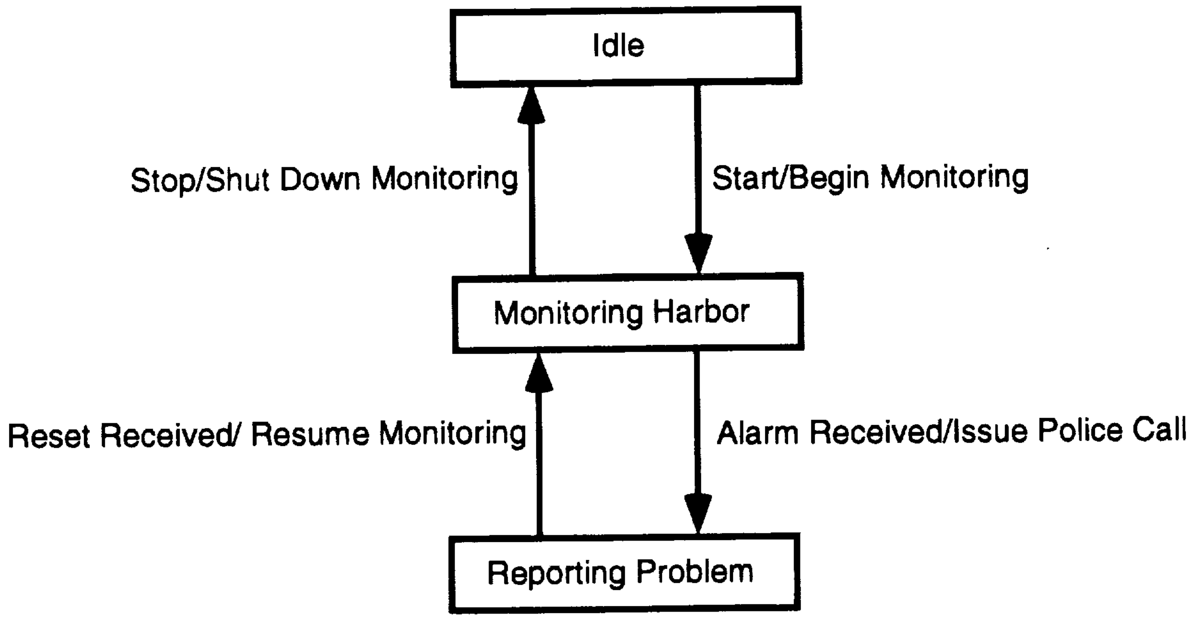

Idle
Stop/Shut Down Monitoring
Start/Begin Monitoring
Monitoring Harbor
Reset Received/ Resume Monitoring
Alarm Received/Issue Police Call
Reporting Problem

Figure 4.1-5 State-Transition Diagram Example

4.1.1.3 State Transition Diagrams

State transition diagrams (STDs) are a part of some structured analysis approaches, for example, real-time structured analysis (RTSA). They are used to specify the sequence and conditions under which processes shown on dataflow diagrams will be operative. There is not a hierarchy of diagrams as with dataflow diagrams. Instead, STDs are packaged with their associated dataflow diagrams by drawing an icon representing the STD on the dataflow diagram and expanding the STD on its own page. State transition diagrams contain graphic symbols representing:

- states (rectangles)
- transitions (directed vectors connecting states).
- condition/action (text annotations on vectors)

Figure 4.1-5 is an example of a state transition diagram that could be packaged with the DFD shown in Figure 4.1-3.

4.1.1.4 Process Specifications

A process specification is a rigorous, non-ambiguous description of the processing that a primitive level function (depicted as one of the circles on the lowest level of DFDs) will perform. It specifies how the process transforms its input dataflows into its output dataflows, outlining whatever data reduction, combination, or calculation algorithms are to be used. The specification can be in the form of a bullet list, decision tree, structured English, or pseudo-code. It should be at least half a page, but no more than one full page in length.

4.1.1.5 Data Dictionary

The data dictionary defines data flows in terms of their constituent data elements and defines data elements in terms of their description. The data dictionary uses a notational syntax consisting of the following symbols:

- = the item to the left is defined as consisting of the components on the right;
- + the two adjoining items are components of the item being defined;
- (x) item x is optional within the dataflow;
- {x} there are multiple occurrences of item x within the dataflow;
- */x/* x is a free form comment or description.

The following is a sample data dictionary dataflow definition:

Duty_Roster = {Guard_Name + Guard_Number + {Harbor_Name + Harbor_Number + Time_In + Time_Out}}

4.1.1.6 CASE Tools for Structured Analysis

The acronym CASE stands for Computer-Aided Systems Engineering and represents a host of commercial application software designed to provide automated support for requirements analysis and design specification. The tools allow the user to draw diagrams such as those shown above and enter associated specifications such as the process specifications and data dictionary definitions referred to above. Embedded in such tools is knowledge of the methodology(s) being supported so that users can check models and specifications for consistency with methodology guidelines. This is a way of getting an automated inspection (see 5.4.1, Inspections). Structured analysis methodologies are well supported by many CASE tools operating on personal computers and workstations.

4.1.2 Structured Design

The SEPG supports an approach to structured design usually referred to as Yourdon/Constantine and also supports the Ward/Mellor approach to real-time structured design. These are probably the most widely used structured design approaches. Several fairly current references are excellent for learning and applying this approach, in particular see [CONST] and [WARD].

4.1.2.1 Physical Dataflow Diagrams

Physical dataflow diagrams provide a means for bridging structured analysis to structured design. Assuming that the goal of a structured design effort is to represent the proposed software architecture as a hierarchy of software modules, the graphic representation of this hierarchy might be a structure chart (see Figure 4.1-6). But how can structured analysis dataflow diagrams be translated into structure charts?

A software module is an *atomic* (that is, not divisible) piece of executable software. Structured design principles say that this unit should be as small as possible; no more than 200 lines of code. Some projects might set standards for smaller units than this and some projects might allow for larger units to optimize performance on supercomputers. Physical dataflow diagrams are a way of organizing these modules into a functional hierarchy that provides functionality equivalent to that modeled in the analysis phase. Because each primitive process now represents one or more actual software modules (depending on the size of the modules), some repartitioning is necessary. This may result in everything below the context dataflow diagram changing. The resulting set of physical dataflow diagrams is usually referred to as the code organization model.

4.1.2.2 Physical Entity Relationship Diagrams

Whereas the logical Entity Relationship Diagram (ERD) might be a model of the composition of a data store on a dataflow diagram, the structured design ERD provides a model of an actual database or related file collections.

4.1.2.3 Structure Charts

Structure charts are a part of all structured design approaches. They are used to specify the actual software modules in a proposed software application system. Their procedural dependencies, as with dataflow diagrams, include order of execution, passed parameters, and runtime flags. There is a hierarchy of diagrams; structure charts contain graphic symbols representing:

- software modules (boxes)
- procedural dependencies (directed vectors connecting modules)
- parameters and flags (labeled mini vectors annotating procedural dependencies)

Figure 4.1-6 is an example of a structure chart.

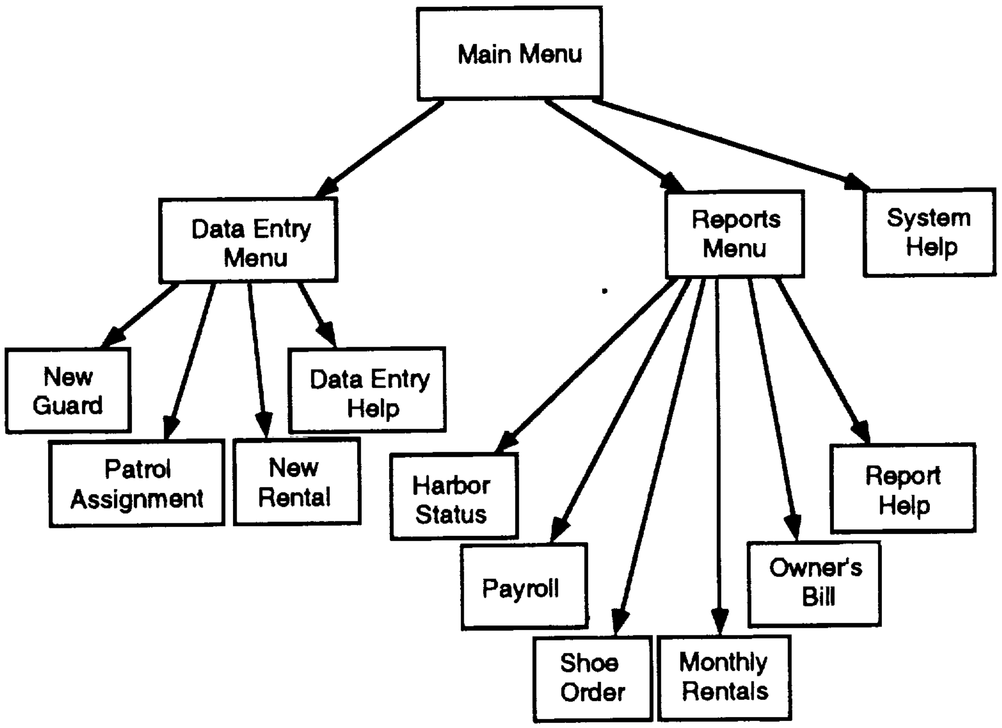


Figure 4.1-6 Structure Chart Example

4.1.2.4 Module Specifications

A module specification is very similar to a process specification, but is associated with a module on a structure chart rather than a process on a DFD. It is a program level specification and should provide sufficient detail for coding.

4.1.2.5 CASE Tools for Structured Design

These are usually a module within a CASE tool that also supports structured analysis.

4.1.3 Structured Programming

Structured Programming was introduced in the mid-1960's and carried forward into most commercial programming books. For specifics, see almost any book on programming languages.

4.2 Object-Oriented Development

If the software developer chooses to apply object-oriented techniques to the software development process, then the appropriate mapping of techniques to phases in Figure 4.2-1 is as follows:

Phase	Technique
Requirements Definition/Analysis	Object-Oriented Analysis & Inspections
Software Preliminary Design	Object-Oriented Design & Inspections
Software Detailed Design	Object-Oriented Design, Testing & Inspections
Implementation	Object-Oriented Programming, Testing & Inspections
System Integration	Object-Oriented Testing
Installation & Acceptance Testing	Object-Oriented Testing

These techniques are described in paragraphs 4.2.1 through 4.2.3.

4.2.1 Object-Oriented Analysis

The SEPG supports an approach to object-oriented analysis usually referred to as Coad/Yourdon. It is important to note that acceptable variants also exist, such as Schlaer and Mellors' Object-Oriented Systems Analysis [SCHLAER]. For an excellent reference for learning and applying the Coad/Yourdon approach, see [COAD-OOA].

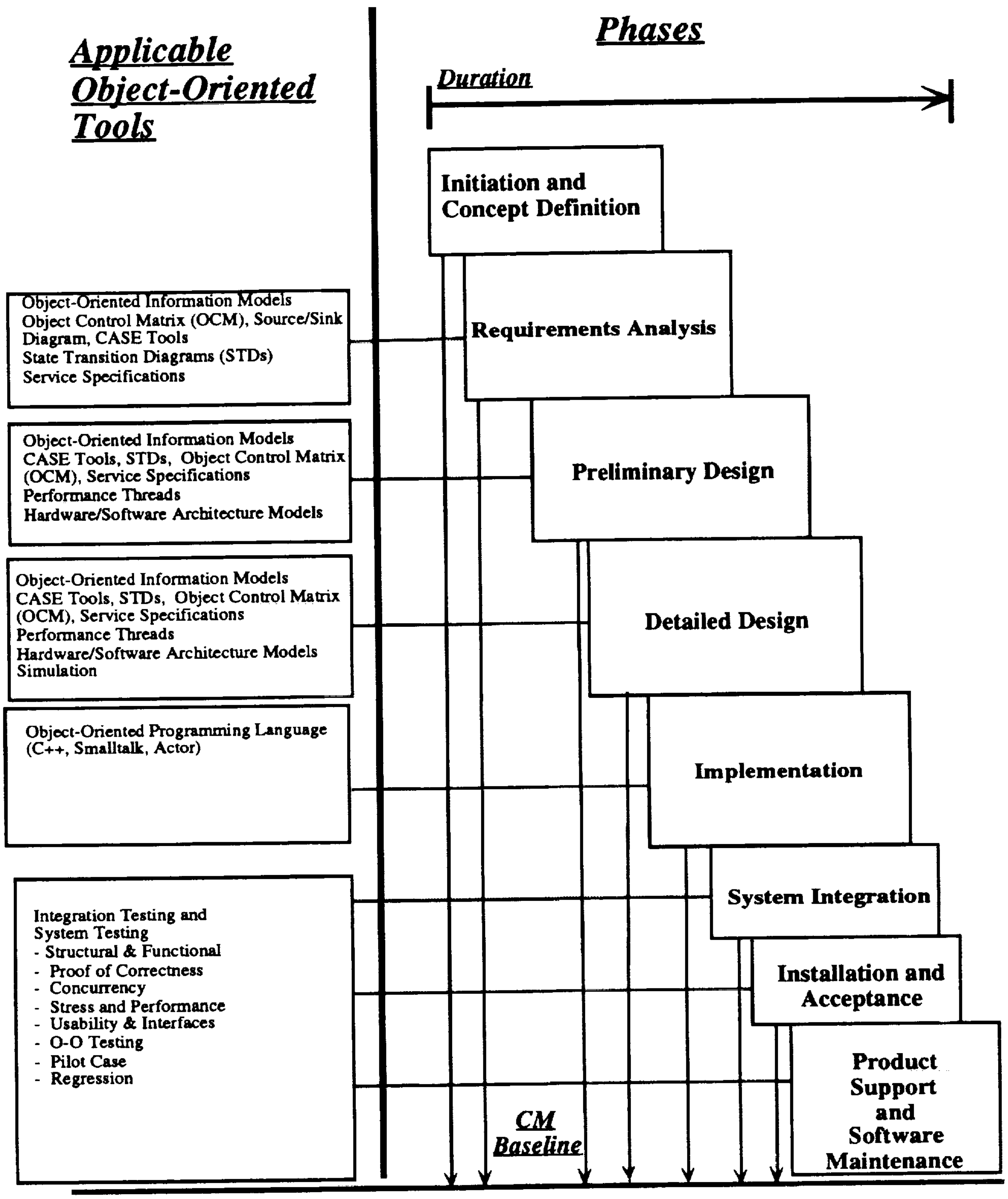

Applicable Object-Oriented Tools
Phases
Duration
Initiation and Concept Definition
Requirements Analysis
Preliminary Design
Detailed Design
Implementation
System Integration
Installation and Acceptance
Product Support and Software Maintenance
Object-Oriented Information Models
Object Control Matrix (OCM), Source/Sink Diagram, CASE Tools
State Transition Diagrams (STDs)
Service Specifications
Object-Oriented Information Models
CASE Tools, STDs, Object Control Matrix (OCM), Service Specifications
Performance Threads
Hardware/Software Architecture Models
Object-Oriented Information Models
CASE Tools, STDs, Object Control Matrix (OCM), Service Specifications
Performance Threads
Hardware/Software Architecture Models
Simulation
Object-Oriented Programming Language (C++, Smalltalk, Actor)
Integration Testing and System Testing
- Structural & Functional
- Proof of Correctness
- Concurrency
- Stress and Performance
- Usability & Interfaces
- O-O Testing
- Pilot Case
- Regression
CM Baseline

Figure 4.2-1 Object-Oriented Development Process Phases and Tools

4.2.1.1 Object-Oriented Information Models

The Object-Oriented Information Model is the central model of the Coad/Yourdon object-oriented analysis approach [COAD-OOA]. It is used to define:

- application object classes;
- attributes of each object class;
- services to be provided by each object class;
- relationships between object classes;
- inheritance structures;
- subject layering (domain analysis).

There is no hierarchical decomposition, other than subject layering, for these diagrams. Each object class has multiple instances, identified by one or more unique attribute values. (A software object is an abstraction of a real-world thing of interest to a user—defined by the object's attributes and computerized behavior that are of interest to the user.) Objects can be: other systems, devices, events, document components, persons in roles, procedures, sites, and organizational units. Objects have attributes that need to be remembered by the system and they require services such as create, modify, delete, monitor, and calculate. The Object-Oriented Information Model contains graphic symbols representing:

- object classes (rounded rectangles with the class name at the top)
- class attributes (a list in the middle of each object class)
- services (a list at the bottom of each object class)
- relationships — called "instance connections" (lines connecting classes)
- cardinality (notations such as 1,m at the end of a line)
- inheritance (a semicircle connecting a set of classes to their parent class)
- whole-part structures (a triangle connecting a set of classes to their parent class)
- subject areas (boundary lines around a collection of classes)

Figure 4.2-2 is an example of an Object-Oriented Information Model.

4.2.1.2 Object Control Matrices

An Object Control Matrix (OCM) is not a part the Coad/Yourdon object-oriented analysis approach. It is described in *Object-Oriented Rapid Prototyping* [CONN91]. It is used to specify the events (messages) that will cause specific object classes to provide specified services. The Object Control Matrix has a row for each object class labeled with the name of the class. The matrix associates each service with the interception of a message by an object.

Due to the nature of object-oriented environments, messages proceed up the object class inheritance hierarchies. A message may be sent (by an event such as a user action) to a subclass, but the service might be provided by a higher level class. Some messages may trigger different services, depending on the responding object, and some messages may trigger the same service (for example, create) from a set of responding objects.

Figure 4.2-3 is an example of an Object Control Matrix.

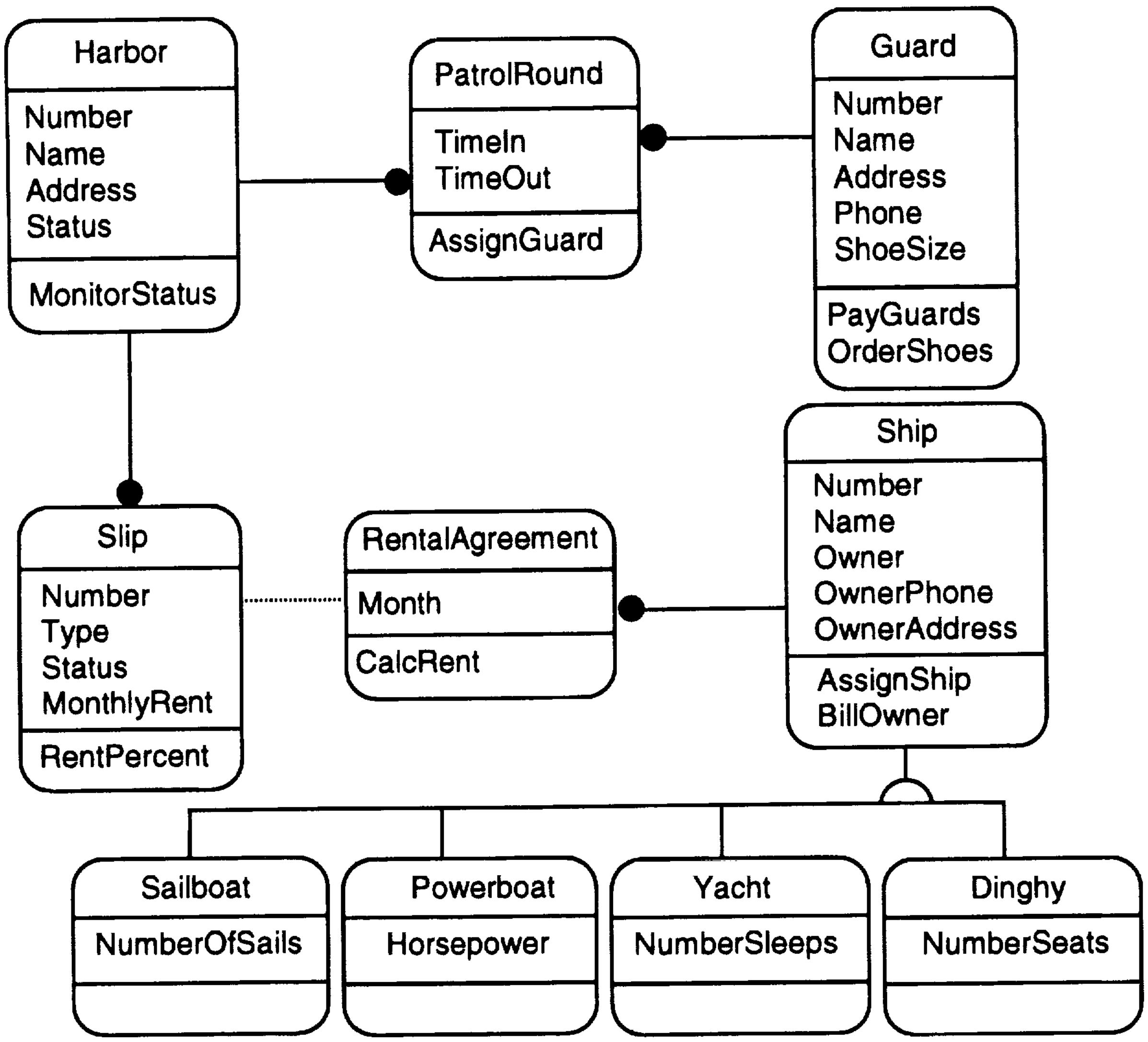


Figure 4.2-2 Object-Oriented Information Model Example

	MESSAGES			
OBJECTS	Payday	MonthEnd	New	EndContract
Guard	PayGuards		OrderShoes	
RentalAgrmt		CalcRent	CalcRent	CalcRent
Ship		BillOwner	AssignShip	BillOwner
Slip		RentPrcnt		

Figure 4.2-3 Object Control Matrix Example

4.2.1.3 Object Source/Sink Diagrams

The Object Source/Sink Diagram shows where application data comes from (external sources) and where data is delivered to (external destinations) outside the application domain. This diagram provides an initial means for modeling critical application objects as the information repositories necessary for collecting and distributing information to and from external entities.

The object source/sink diagram is not part the Coad/Yourdon object-oriented analysis approach. It is only recommended here as an aid in getting started with an object-oriented analysis. The analyst will ask users what data the system must be capable of producing (displays, reports, database and file updates, device control) and which people, files, and devices are to receive the data. The analyst also should ask which people, files, and devices the data will be obtained from and what the components of the data are, then derive a set of problem domain object classes which serve as repositories for the incoming and/or outgoing data. The objects will be abstractions of real-world things, based on attributes which match the components of specified dataflows.

The application domain within the source/sink diagram can be treated as a black box; given the specified inputs, the application objects can be assumed to contain services capable of providing the specified outputs. Relationships or instance connections, messages, and services need not be shown on this diagram. A source or sink can be either a user, a device, or an external data store. A single external can be both a source for certain dataflows and a sink for others. A single object class may receive several input dataflows of a particular (related) type.

The source/sink diagram need not be maintained as development progresses; it is only a starting point. However, it may always be useful to have a specification of the system's mission in terms of net dataflow. In any case, do not attempt functional decomposition of the object source/sink diagram. Objects are reusable; the mid-level pieces of a functional decomposition are not.

Object source/sink diagrams contain graphic symbols representing:

- external information sources and destinations (rectangles)
- object classes (rounded rectangles with the class name at the top)
- data flowing from source to destination (directed vectors between externals and internal objects)

Figure 4.2-4 is an example of an object source/sink diagram.

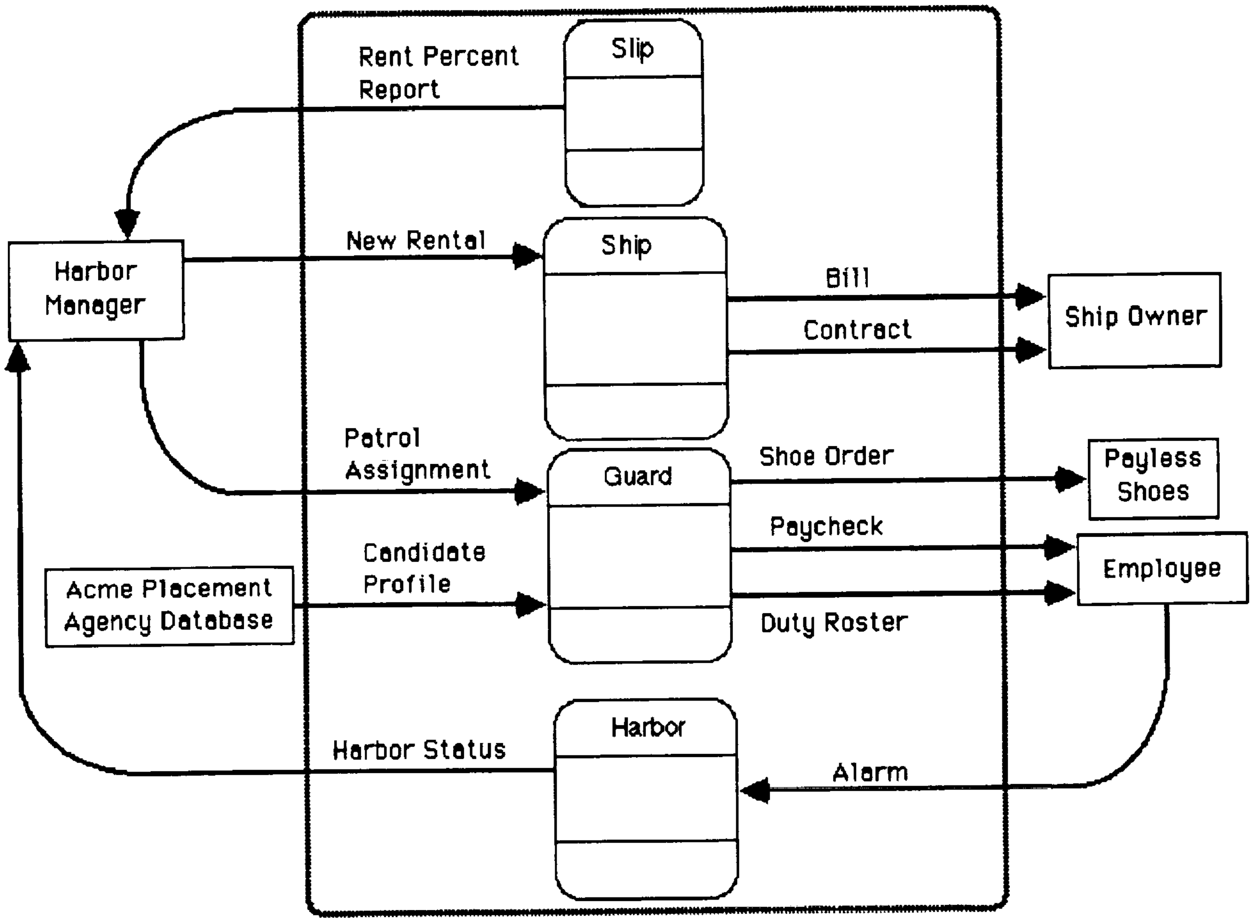


Figure 4.2-4 Object Source/Sink Diagram Example

4.2.1.4 Object State Transition Diagrams

State Transition Diagrams (STDs) are not a required part of the Coad/Yourdon object-oriented analysis approach, but their OOA text has an excellent explanation of how to apply them in an object-oriented requirements specification [COAD-OOA]. They are more commonly seen as part of the Schlaer/Mellor [SCHLAER] object-oriented analysis approach and look exactly like the diagrams by the same name from the Ward/Mellor real-time structured analysis (RTSA) approach [WARD]. They may be used in OOA to specify the sequence and conditions under which services of an object class will operate.

4.2.1.5 Service Specifications

Services containing computational complexity or multiple message handlers will need an additional program unit graphic model. Coad/Yourdon approach suggests a flow chart in the second edition, but suggests a data flow diagram fragment in the first edition. Either is acceptable.

Figure 4.2-5 is an example of a data flow diagram fragment used as a service specification program unit graphic.

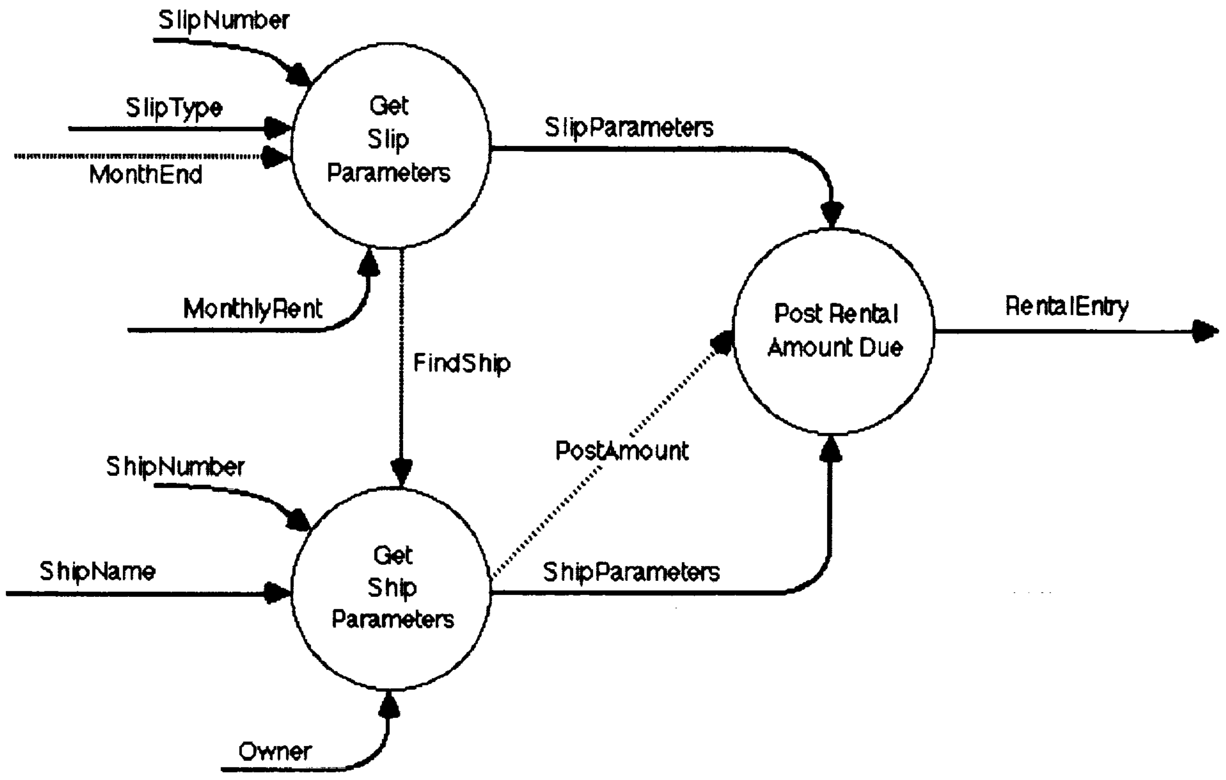


Figure 4.2-5 DFD as a Service Specification Example

4.2.1.6 CASE Tools for Object-Oriented Analysis

Any CASE tool supporting Information Modeling and Data Dictionary Management with Process Modeling can be used effectively. Some examples of such tools include: IDE's Software Through Pictures, CADRE's Teamwork, Sybase/SQL Solution's Deft, and Popkin's Software Architect. The tool does not have to specifically support the object-oriented methodology being used. In fact, since methodologies usually change first, and then the tools play catch-up, CASE tools rarely support methodologies in an exact manner.

4.2.2 Object-Oriented Design

The SEPG supports the Coad/Yourdon approach to object-oriented design. It is important to note that acceptable variants also exist, such as Schlaer/Mellor. For an excellent reference to learn and apply the Coad/Yourdon approach, see [COAD-OOD].

The Coad/Yourdon approach to object-oriented design is basically to build on the OOA models (do more of the same) with a particular physical implementation in mind. The same models are used to produce implementation specific representations of the components of the system architecture.

4.2.3 Object-Oriented Programming

Object-oriented programming is done with a language that supports one or more of the following concepts:

- Message passing - the ability to generate and automatically trap messages from within services;

- Inheritance - the ability to create new objects by building on the definition of old objects, with changes to the parent object being automatically inherited by the child;
- Encapsulation - creates software objects that have everything they need (data and functions) to provide required services;
- Polymorphism - each object can interpret a given message in its own way.

There are three types of object-oriented programming languages: pure, hybrid, and quasi object–oriented. Pure object-oriented languages were originally developed as such, and support all the above concepts. Examples of such languages are Smalltalk, Actor, and Eiffel. Hybrid object-oriented languages also support all of the above concepts, but do so through extensions to the compiler of a conventional procedural language such as C or Pascal. Thus, examples of hybrid object-oriented languages are C++ and Object Pascal. Quasi object-oriented languages only support a subset of the above concepts. Examples of quasi object-oriented languages include Ada and HyperTalk.

4.3 Evolutionary Rapid Prototyping Development

Because the term rapid prototyping is subject to so many different definitions and interpretations (frequently misinterpretations), the SEPG supported definition of rapid prototyping is covered separately, in paragraph 4.4, "Prototyping." The following paragraphs cover the appropriate tools for rapid prototyping.

4.3.1 Use of Structured or Object-Oriented Methods

When rapid prototyping is performed on a project, it should be used to augment requirements and design specifications, not to replace them. Either structured analysis and structured design or object-oriented analysis and object-oriented design can and should be performed concurrently with prototype iteration and refinement to produce high quality specifications. The graphic models described above should be used to specify the initial prototype and then expanded along with the prototype over many iterations based on user feedback from prototype demonstrations.

If object-oriented analysis and design are being used, the initial prototype specification will consist of a source/sink diagram, object-oriented information model, and object control matrix (a three-page specification). If structured analysis and design are being used, the initial prototype specification would consist of a context level dataflow diagram, essential functions dataflow diagram, entity-relationship diagram, and structure chart (a four-page specification). In either case the specification of the initial prototype should consist of only a fraction of the final product (perhaps 5 to 20%). This is to avoid prespecification, since the point of prototyping is to discover requirements. At the end of prototype iteration, however, there should be no difference between the quantity and quality of these specifications and similar specifications produced in a sequential build in a conventional life-cycle.

4.3.2 Very High Level Development Environments

Rapid Prototyping is, by nature, a tool dependent approach. Very high level development environments are required to ensure successful rapid prototyping. Needed are prototype development tools that allow for an integrated approach to the prototyping of data structures, functionality, and user interface requirements. The tools also must allow for the capture and output of live data with which the user is familiar. These tools must be capable of creating software that will be just as easy to modify as the analysis specifications. If not, the prototype iteration process will take too long and the software will become difficult to maintain due to poorly designed modifications during iterations. The best prototyping tools will allow a user-approved prototype to evolve into an easy-to-maintain production system.

If conventional programming languages are used for prototyping, it will not be possible to rapidly modify the prototype, based on user feedback, over dozens of iterations, while concurrently refining and expanding the specifications. Conventional software, written with third generation procedural languages, is too difficult to modify. High quality prototyping (with adequate specifications resulting in maintainable code) will not be cost effective with such a language. Examples of poor prototyping languages include: COBOL, FORTRAN, C, Pascal, Ada, Basic, and Lisp. The object-oriented extensions to some of these languages, such as C++, ObjectPascal, and CLOS make them acceptable prototyping environments in certain situations, given robust, heavily populated class libraries.

Good prototyping environments make the prototype as easy to modify as the requirements and design specifications. To accomplish this, the prototyping environment must include:

- a very high-level (fourth generation or better), declarative, non-procedural programming language;
- visual programming tools to develop user interface components without coding;
- a flexible data management system where schema changes are easy to accomplish;
- application integration tools to allow execution control without job control language.

Often, such a tool suite can be used for effective prototyping, even if the code has to be completely rewritten after requirements are discovered and baselined. This throwaway approach is better than developing a difficult to modify prototype with a conventional programming language.

Figure 4.3.2-1 shows how some current development tools might be evaluated as candidate rapid prototyping tools in light of these criteria. This is not a complete list, but rather, simply a sample of representative tools with which a SEPG are familiar and for which, support, in terms of training and consulting, could be made available to projects. The intent behind providing Figure 4.3.2-1 is not to endorse or condemn specific products, but rather to illustrate how criteria for rapid prototyping development environments can be applied to the evaluation of specific commercial products as a part of task planning. The ratings given in Figure 4.3.2-1 are primarily examples. Each of the categories presented should be considered, but of course, more may be added for a finer degree of environment-specific ratings. Evaluations based on hands-on product testing within a specific environment are recommended.

There are tradeoffs to be considered with any tool. Relational database management systems, although not originally designed as such, have proven very effective as rapid prototyping tools. Languages, such as C++, produce very reusable software, but are perhaps not such good prototyping tools because of their tendency to produce software that is more complex and thus more difficult to modify than the scripts of a higher level language such as HyperCard's HyperTalk. One of the most serious difficulties of rapid prototyping is finding excellent rapid prototyping tools that also produce optimally reusable software. Software produced with expensive proprietary tools is not as reusable as software produced with commonly available compilers.

Another difficulty in publishing a list of supported rapid prototyping tools is that the list is obsolete as soon as it is created. There are literally hundreds of such products, and dozens of new ones are introduced every year. Chances are the ones that were introduced "last week", (ones a SEPG are probably not aware of yet), are considerably more powerful than any on the following list. The tools on this list have the following advantages:

- An experience-based evaluation of their merit for prototyping;
- A means to assess the reuse tradeoff;
- A multi-project local user base;
- SEPG will provide some level of support, if appropriate.

The criteria for including tools on this list are:

- At least two projects locally are using the product;
- The product is being used successfully on at least one local project to support a formalized approach to structured or object-oriented rapid prototyping as described in this guidebook;
- At least one member of the SEPG is technically familiar with the product or is working closely with someone locally who is.

Of course, any tool purchased must be selected by a competitive procurement based on project requirements.

Product (Company)	GUI Development Speed	Modifiable Data Structure	Function Development Speed	Reuse Rating
Sybase (Sybase)	☆☆	☆☆☆☆	☆☆	☆
4th Dim. (ACI)	☆☆☆☆	☆☆☆☆	☆☆☆☆☆	☆☆
HyperCard (Apple)	☆☆☆☆☆	☆☆☆☆	☆☆☆☆☆	☆☆☆☆
Labview (Nat. Instruments)	☆☆☆☆☆	☆☆	☆☆☆☆	☆☆☆
C ++	☆	☆	☆	☆☆☆☆☆
TAE (NASA)	☆☆☆	☆	☆	☆☆☆☆
JAM (JYACC)	☆☆☆☆☆	☆☆☆☆	☆☆☆☆	☆☆☆

Figure 4.3.2-1 SEPG-Supported Rapid Prototyping Development Tools

4.3.3 Reuse in Prototyping

Reuse is the third type of tool useful for rapid prototyping. A software library, filled with reusable code, is a powerful prototyping tool. Reusing existing code is even faster than development with a very high level development environment. Effective reuse is, however, subject to the following conditions:

- limited reusability of conventional, procedural language software programs;
- conscious design for reuse;
- decreased maintainability of modified procedural language code;
- ease of locating reusable components.

The last condition is usually the most difficult, but is possible with good reusable software library management systems and good help desk staffing. Again, the prototyper should perform preliminary analysis and design first, then search the reuse libraries for software that matches design components. If object-oriented analysis and design are used, this task will be easier, and any resulting developed software will be easier for future developers to reuse.

Conventional programs will be more difficult to reuse than object-oriented software. Objects are encapsulated (minimizing the ripple effect of changes), can be extended easily through inheritance, and are independent of control architecture because of messages and polymorphism. Conventional programs are built to exist within a specific functional hierarchy and may have difficulty being reused outside that environment. If such software requires extensive rework, as determined from user feedback at prototype demonstrations, it is advisable to rewrite the software in a higher level language before proceeding with prototype iteration. Otherwise, the project will become too expensive, and the resulting software may not be easy to maintain.

4.4 Prototyping

The following paragraphs define the approach to use when rapid prototyping is selected for a software development project. The tools to be used when applying this technique are shown in Figure 4.4-1.

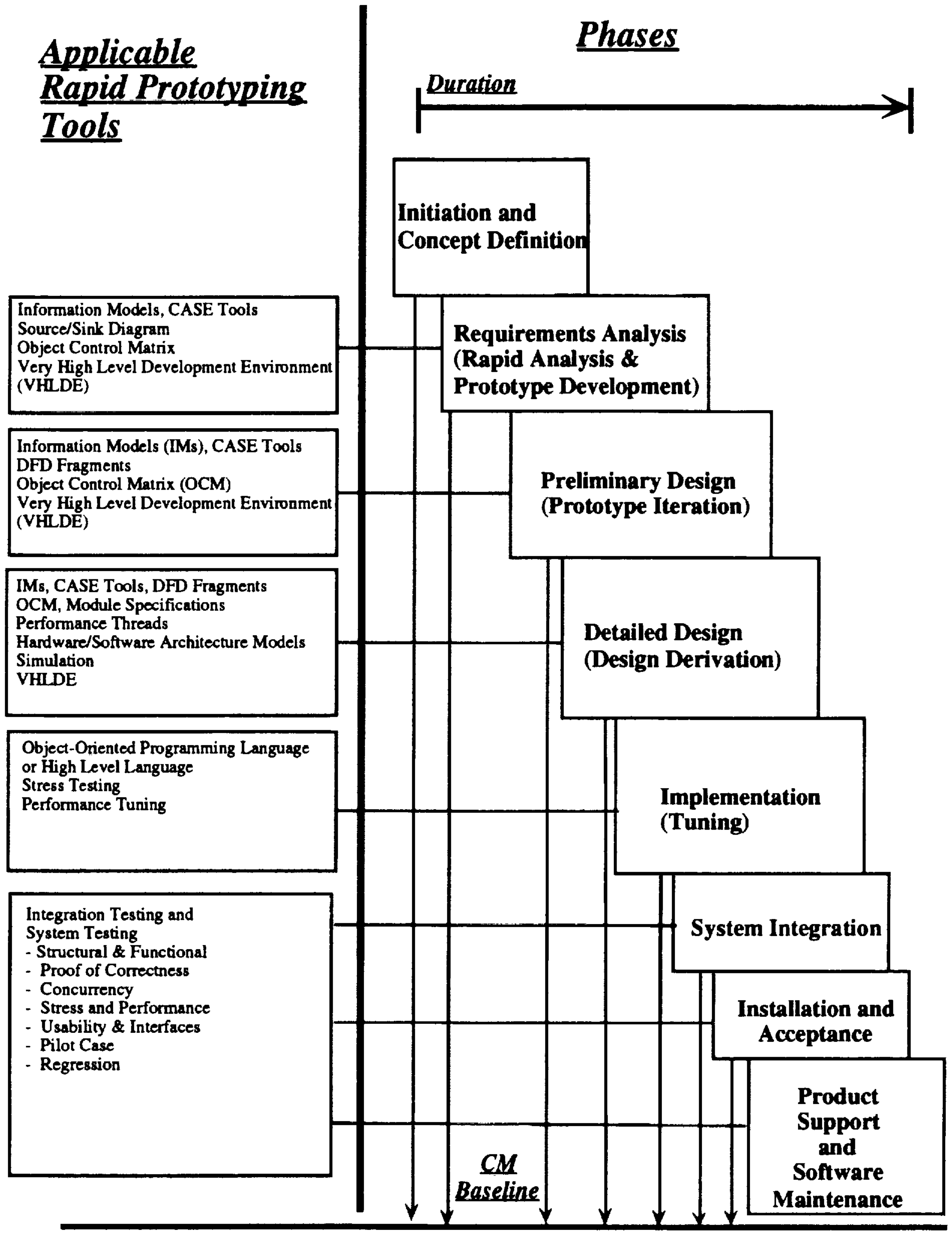

Applicable Rapid Prototyping Tools
Phases
Duration
Initiation and Concept Definition
Information Models, CASE Tools
Source/Sink Diagram
Object Control Matrix
Very High Level Development Environment (VHLDE)
Requirements Analysis (Rapid Analysis & Prototype Development)
Information Models (IMs), CASE Tools
DFD Fragments
Object Control Matrix (OCM)
Very High Level Development Environment (VHLDE)
Preliminary Design (Prototype Iteration)
IMs, CASE Tools, DFD Fragments
OCM, Module Specifications
Performance Threads
Hardware/Software Architecture Models
Simulation
VHLDE
Detailed Design (Design Derivation)
Object-Oriented Programming Language or High Level Language
Stress Testing
Performance Tuning
Implementation (Tuning)
Integration Testing and System Testing
- Structural & Functional
- Proof of Correctness
- Concurrency
- Stress and Performance
- Usability & Interfaces
- Pilot Case
- Regression
System Integration
Installation and Acceptance
Product Support and Software Maintenance
CM Baseline

Figure 4.4-1 Evolutionary Rapid Prototyping Process Phases and Tools

4.4.1 Purpose and Objectives

Rapid prototyping is a requirements discovery technique. It is useful for discovering valid data, control, processing, and performance requirements. A rapid prototype usually will not validate requirements; it will tend to demonstrate where prespecified requirements are invalid. Users' opinions about the validity of requirements implementation, as demonstrated by the prototype, are incorporated into iterative refinements.

One objective of prototyping is to obtain feedback from users on the appearance of the user interface and system output. Another objective is to define the detailed requirements for data storage, processing functionality, and system control architecture. Lower level objectives can include discovering performance problems and reducing development costs. When structured or object-oriented rapid prototyping is used as a formal method for developing deliverable software in an evolutionary manner, development costs can be significantly reduced (*e.g.*, 10%) and overall life-cycle costs can be dramatically reduced (*e.g.*, 40%) due to reduction of rework during test and maintenance phases [CONN89].

4.4.2 Products and By-Products

All of the normal products of a conventional software development project are produced during a rapid prototyping project. There will still be a requirements specification, design document, test plan, and user's guide. Only the timing of the delivery of these products will be altered by rapid prototyping.

A small preliminary version of the requirements specification will be produced prior to initial prototype development. This document will include the design components necessary to build the initial prototype. When the latest version of the prototype is approved by users, the requirements will be complete. They can be separated from design components and published as a baseline requirements specification. The design components, as prototyped, become the preliminary architectural design specification. A final detailed design specification emerges as the approved prototype is tuned for performance and evolves into deliverable software.

The user guide can be built from on-line interactive help screens, prototyped to evaluate their helpfulness. Thus, development of the user's guide should begin during the earliest stage of prototype development and should be complete at the end of the requirements definition phase of the project. Likewise, test plans and procedures should be completed during requirements definition. Each prototype demonstration should be orchestrated by its own mini test plan which amounts to a script for the demo. The accumulated set of demo scripts for all prototype iterations can then be distilled into a test procedures document at the end of requirements definition.

4.4.3 Risk Mitigation

Rapid prototyping is the best mitigation technique for the following risks:

- Users may not be able to understand how to use the system without prohibitively expensive training;
- The system may not provide the information or services needed to support users in their jobs;
- The system may not perform its processing functions correctly;
- Requirements may become obsolete before they can be implemented;
- The system "look and feel" may not be attractive enough to lure users away from alternatives and, therefore, won't get used;
- The system may be excessively expensive to maintain.

There is no risk associated with formalized rapid prototyping when it is undertaken using the kinds of tools, techniques, and procedures described here.

4.4.4 Prototyping Characteristics and Methods

A good prototype should be fully functional (no display-only services) and should incorporate test data from the user's actual work environment. The user must be able to verify whether prototype services produce accurate, useful results. This verification must be based on an analysis of familiar data after transformations have been performed.

Users have much more active involvement in a prototyping project than they do in a conventional project — often contributing up to 50% of the total development effort. The rapid prototyper always takes the approach that the user interface is the most critical part of an application, even if this will not be the case in the final implementation (for example, embedded software). Even autonomously functioning software will have a requirements commissioner and this person(s) will be the prototype user.

A good prototype should be capable of evolving to the final software product. Whether it actually will or not is another matter, but the prototype should be built with tools that at least allow for this possibility. If good prototyping tools are used, the prototype will always be as easy to modify as the analysis and design specifications. The solution to performance problems is deferred until after requirements definition is finalized.

For the initial prototype, analysis and design specifications are intentionally left incomplete and it is acceptable if they are ultimately determined to be incorrect. The preliminary analysis and design for the initial prototype of even the biggest system should take no more than a month to complete, and for a medium size system this activity should take about one or two weeks. Building the initial prototype, using the right tools, will take only about as much time as the specifications development did.

All life-cycle activities are performed concurrently during the requirements definition, using prototype iteration. Analysis and design specifications are updated during each iteration. Test planning is performed prior to each prototype demonstration and the demonstration itself is a test of a fully integrated prototype.

4.4.5 Analysis and Evaluation of Prototyping Results

A prototype is iterated as many times as necessary until service functionality, data attributes, and control structures are determined to be correct by the users. The number of iterations is typically between 12 and 50 depending on application complexity, user difficulty, and efficacy of the prototyping tool. When users finally approve the prototype, the final requirements document may be published.

For each iteration, the prototype is demonstrated to users for the purpose of discovering additional required services, data attributes, and control mechanisms. Existing services, data attributes, and control mechanisms are evaluated for correctness. Beginning with the second prototype demonstration, always verify that the problems discovered during the previous demonstration have been corrected.

Users have distinct responsibilities during prototype iterations including:

- attend regular prototype demonstrations;
- study output to find errors;

- provide constructive criticism and suggestions;
- identify additional requirements;
- cooperate to converge on a requirements solution within schedule and budget.

If user's won't commit to this level of participation, rapid prototyping may not be advisable. An alternate is to have other people pretend to be users and take the user's point of view, but this will not be as satisfactory as real user involvement.

Development should not continue past prototype iteration until users have expressly approved prototype correctness, completeness, and exactness. At this point, documentation should provide everything needed for conventional software maintenance. An inspection team, staffed by maintenance personnel, should verify that this is true. Existing performance problems should be well documented and preliminary performance requirements specified. Performance problem solutions are developed after the requirements definition is complete.

4.5 Tools not Specific to a Methodology

TBD.

4.5.1 Program Design Languages

TBD.

4.5.2 Performance Thread Analysis

TBD.

4.5.3 Hardware/Software Architecture Models

TBD.

Section 5

ASSURANCE

5.1 Overview

A software error is present when the software does not do what the user reasonably expects it to do. A software failure is an occurrence of a software error. *Glenford Meyers, Software Reliability, 1976* [MEYER]

Assurance is a blend of activities performed throughout the development life-cycle to ensure the creation of correct and appropriate products that accomplish desired results. To be most effective, assurance activities must be an integral part of the development; they cannot be done only at the end of development. Assurance responsibilities span several subgroups of the development effort such as the developers, development management, the customer, and (possibly independent) quality assurance or test groups.

Several major activities compose the assurance area: Risk Analysis, Quality Assurance (QA), Verification and Validation (V&V), and Problem Reporting. Risk Analysis is identifying and analyzing factors that can jeopardize the success of the project or product's use. QA is establishing and enforcing conformance to standards, procedures, and plans. Verification is ensuring that requirements and objectives of the product are correct, complete and that throughout the life-cycle, each development step builds upon and implements what was developed or required in the previous step, "the product was built right". Validation is checking to ensure the product satisfies the requirements, "the right product was built". Problem Reporting is the mechanism for reporting and tracking resolution of errors and problems. Problem reporting is closely related to configuration management. This section addresses the above assurance activities.

Two additional areas of assurance are Quality Engineering (QE) and Quality Audits. Quality Engineering ensures that metric quality indicators (such as reliability and maintainability factors) are set as product requirements and met by the development. Further QE responsibilities are to examine and analyze project metrics and utilize the information to improve the project, product and process procedures. Quality audits are reviews to ensure that standards, processes, and procedures are followed and effective. Audits are described in the Software Quality Assurance Audits Guidebook, see [SQAG]. Quality engineering and audits are not discussed further in this guidebook.

Assurance has two primary concerns in achieving the goal of producing correctly functioning, appropriate products for a customer. These are the "engineering in" of quality (goodness) and to find and remove errors (evil). All assurance activities (and all development activities) are oriented toward achieving the goal and satisfying these concerns.

The most cost effective and efficient methods for finding and removing errors in top-down sequential development are technical reviews and Formal Inspections. The primary cost saving is in finding and fixing errors early in the development, which prevents error compounding. Prototyping holds the possibility for even greater cost savings due to the increased user involvement and discovery of the true requirements early on. Whichever assurance and development approaches are selected, emphasis should be on early definition of accurate requirements and early detection of errors of both omission and commission.

The contents of this section are organized as follows:

- Risk Analysis - identifies risk factors and measures to reduce risk.
- Standards and Guidelines - set and measure conformance to standards for processes and products.
- Reviews - are human-based examination of products for adherence to standards, requirements, and other factors. Reviews are a mechanism used by both QA and V&V to achieve their goals.
- Testing - is machine and human-based exercise of the product for requirements satisfaction, ease of use, and error detection. Testing is primarily a V&V activity.
- Problem Reporting - is recording errors and ensuring that they are repaired.

5.2 Risk Analysis and Management

Risk analysis is a management activity used to identify factors which can jeopardize the success of the project (or can result from the use, misuse or failure of the product) and an estimation of the cost of that failure. Based on the risk factors and their importance, actions must be taken to reduce or mitigate the risk. Risks to any development are the required product rework, decreased operational capability or the increased operational cost due to errors from improper development methods and/or inadequate assurance and management methods. This guidebook provides several supported methods which can help reduce some of the development risks for a software project.

Both NASA Headquarters and Ames require that software development (or acquisition) projects be classified by their risk factors and that management address those risks through appropriate management, assurance, and documentation activities. The diagram below is taken from Ames Handbook *(AHB) 5333.1, Establishment of Software Assurance Programs,* and provides a matrix from which to determine a project's risk level. For a project's overall risk level (from Minimum to High), AHB 5333.1 also describes required Management Plan sections, documents to be delivered, and specific assurance reviews which must be held. Use of the Risk Assessment Matrix is described in AHB 5333.1.

	Ames Risk Categories			
Project Risk Factor	1. HIGH	2. MEDIUM	3. LOW	4. MINIMUM
Human	Death	Severe Injury	Minor Injury	Mental Stress
Vehicle/ Facility	Total Loss	Major Damage	Moderate Damage	Minor Damage
Software Acquisition Cost	>$20.0M	$2.5M - 20.0M	$100K - 2.5M	$25K - 100K
Technology	Complex/ Leading Edge	Complex/ State-of-the-Art	Complex/ State-of-the-Practice	Simple/ State-of-the-Practice
Visibility	International Visibility	National Visibility	Agency Visibility	Center Visibility

Figure 5.2-1 Ames Risk Level Assessment Matrix

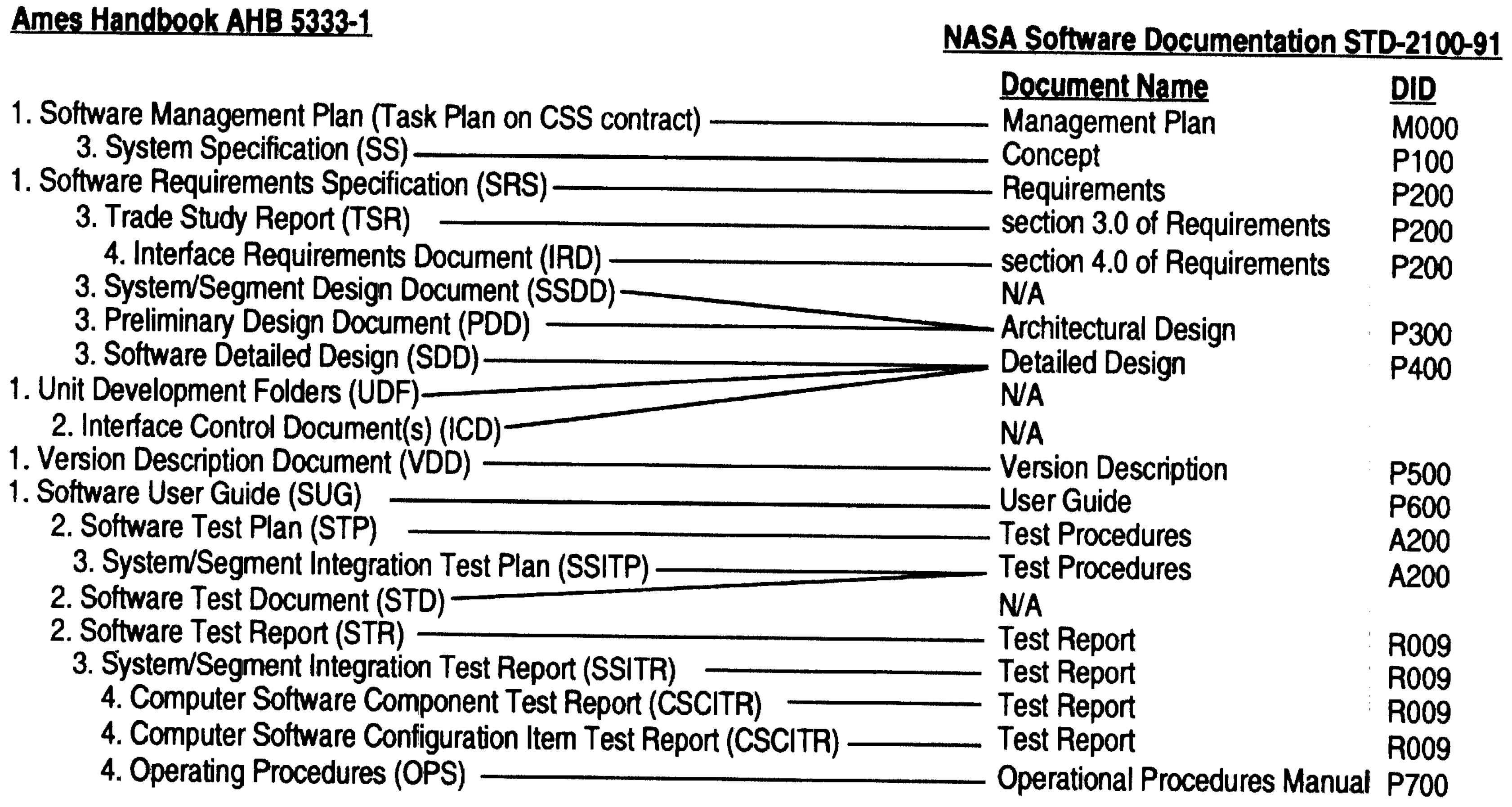
AHB 5333.1 was produced referencing the SMAP 4.3 documentation standards. The figure 5.2-2 maps the AHB referenced documents to the current NASA-STD-2100-91 Note that the Unit Development Folder (UDF) is not described or defined as a DID by the current standards.

Much software produced in the small and medium size NASA research projects fall into the Minimum risk category. By AHB 5333.1, for a Minimum risk project, at least the following actions must be taken:

- Produce a Management Plan, a Software Requirements Specification, a Version Description Document, and a Software User's Guide. *NASA-STD-2100-91 Software Documentation Standards* [DOCSTD] provides an outline for these documents. The Unit Development Folder contents should be included as part of the design documentation.

- Hold a Requirements Review of the requirements document and an Operational Readiness Review of the test results.

The risk category and the management approach used will determine the assurance activities needed and required for the project. Although the following sections describe assurance activities and responsibilities, the initial determination of the appropriate assurance activities is a combined management and assurance responsibility.


Ames Handbook AHB 5333-1
NASA Software Documentation STD-2100-91
Document Name
DID
1. Software Management Plan (Task Plan on CSS contract) — Management Plan M000
3. System Specification (SS) — Concept P100
1. Software Requirements Specification (SRS) — Requirements P200
3. Trade Study Report (TSR) — section 3.0 of Requirements P200
4. Interface Requirements Document (IRD) — section 4.0 of Requirements P200
3. System/Segment Design Document (SSDD) — Architectural Design P300
N/A
3. Preliminary Design Document (PDD) — Architectural Design P300
3. Software Detailed Design (SDD) — Detailed Design P400
1. Unit Development Folders (UDF) — Detailed Design P400
N/A
2. Interface Control Document(s) (ICD) — Detailed Design P400
N/A
1. Version Description Document (VDD) — Version Description P500
1. Software User Guide (SUG) — User Guide P600
2. Software Test Plan (STP) — Test Procedures A200
3. System/Segment Integration Test Plan (SSITP) — Test Procedures A200
2. Software Test Document (STD) — Test Procedures A200
N/A
2. Software Test Report (STR) — Test Report R009
3. System/Segment Integration Test Report (SSITR) — Test Report R009
4. Computer Software Component Test Report (CSCITR) — Test Report R009
4. Computer Software Configuration Item Test Report (CSCITR) — Test Report R009
4. Operating Procedures (OPS) — Operational Procedures Manual P700

- Level 1 = Minimum Risk, Minimum Documentation Set; Each Higher Level Adds Required Documents to Minimum Set.
- STD-2100 Documents are mostly rollouts of the Product Specification, P000.
- 5333-1's "System Specification" really has no STD-2100 equivalent, as the latter is software only.
- The same is true for 5333-1's "System Segment Design Document".
- UDFs and ICDs now have no DID under STD-2100; it is recommended that any important elements of these now obsolete documents be included in the Detailed Design Document. Note: this implies that a Detailed Design should be specified even for Minimum Risk projects.
- The plethora of test documents under 5333-1 is now reduced to Test Procedures and Test Reports.

Figure 5.2-2 AHB-5333.1 Documents to NASA-STD-2100 Mapping

5.3 Standards and Guidelines

Standards and Guidelines provide the ideal to which a product or process is compared. Guidelines provide a hopeful comparison point but standards are required to be met. Various NASA contractors have produced a number of guideline documents to help developers and managers utilize better processes or to help understand and apply NASA standards.

This Guidebook *(Software Engineering Guidebook, 1993)*, provides SEPG approved and supported process and activity guidelines for software development, assurance and configuration management.

NASA Guidebooks described below cover a variety of subjects and are mostly in draft form. The guidebooks provide suggestions and guidance in practical application of NASA standards to projects. Some of the guidebooks address the previous NASA life-cycle and documentation standards [SMAP4.3] while others have been prepared to address the current Documentation Standards [DOCSTD]. However, the essential information is consistent and correct.

Software Assurance Guidebook SMAP-GB-A201, September 1989, [SAG] describes various assurance activities as applicable to NASA developments. This guidebook provides a more detailed and formal description of assurance activities than the section you are currently reading.

Software Formal Inspections Guidebook (Draft), 1990, [SFIG] describes the Inspections process, roles and activities and provides pointers for establishing an Inspections program. .

Software Formal Reviews Guidebook (Draft), 1990, [SFRG] describes formal reviews (including Phase transition reviews) held between the producer and acquirer of software products.

Software Quality Assurance Audits Guidebook (Draft), June 1990, [SQAG] describes audits to check processes against standards, products against requirements, or activity against schedule.

Understanding and Tailoring the NASA Software Documentation Standard (draft), May 1992, [SEI92] explains how to tailor and use the documentation standards, including roll-in and roll-out.

Software Management Plan Guidebook (Draft), May 1992, [SMPG] describes how to meet the Documentation Standards requirement for a management plan, what to include, and how to develop it.

The following standards are applicable to every development project. NASA standards typically contain a clause that says the Program Manager should adapt (or tailor) the standards to their particular project.

NASA Software Engineering and Quality Assurance Standard, DS-XXXX (draft) December 1991, [SWASTD] provides assurance requirements.

NASA Software Documentation Standard, NASA-STD-2100-91, July 1991, [DOCSTD] establishes standard format and content for document deliverables generated during software development or acquisition. The specific document formats are called Data Item Descriptions, known as DIDs.

NASA Software Inspection Process Standard, NASA-STD-XXXX (Draft), March 1992, [SWISTD] describes standards for holding and using Software Formal Inspections.

AHB 5333.1 Establishment of Software Assurance Programs, June 1992, [AHB] provides the Ames local implementation of NASA requirements, see [NMI] for categorizing software risk and setting minimum requirements for documentation, management, and assurance to mitigate that risk.

FIPS (Federal Information Processing Standards) and *FED-STDs* provide a series of government-wide standards for ADP, telecommunications equipment, services, related software requirements (including languages) and terminology related to these areas. Many FIPS and FED-STDs are "pass-through" requirements which adopt IEEE or ANSI industry-wide standards as the government standard. Projects which are not covered by Ames, company or other standards or are implementing technology not covered by locally available standards should adopt appropriate FIPS.

Programming Standards which should be adopted for every project and can be obtained from your company, NASA site, ANSI standard or a commercial source and customized for your specific project.

5.4 Reviews

Reviews are human-based examinations to find problems and deficiencies of software products and processes. Reviews work best when there are standards for content and preparation for the product being. Several types of reviews are available, depending on the review objectives: Software Inspections are peer technical reviews not led by the author of the product being inspected; walkthroughs are reviews generally led by the author; formal reviews are usually a combined contractor/acquirer management review of products, processes, or end-of-phase schedule and progress determination.

Although the review techniques may differ, the primary objective is common: find problems so they can be removed. Other objectives are more functionally oriented depending on the review purpose and the reviewer's purpose; a QA organization will have different objectives from the developers, testers and the Quality Engineering staff. Though differing objectives may be realized in a review there are common minimum requirements to any effective review. These are:

- Standards of comparison for the product should exist (such as programming standards, design representation, and maximum errors per unit) and the standards should be met or exceeded before the material passes that review.
- The level of preparation should be defined and the material properly prepared to that level.
- Review material should be provided to the reviewers prior to the meeting so they can prepare and be familiar with the material. The review should not take place unless all reviewers are prepared.
- Reviewers should be selected who are qualified to contribute. A breadth of reviewers (such as manager, user, implementer, maintainer, tester) should provide different points of view in the review.
- A facilitator should conduct the review flow and maintain the focus.
- The review should be non-threatening to encourage free discussion and revealing of weaknesses that might not be discovered in a hostile environment. The primary purpose is to find errors in the materials reviewed and not in the author of the materials.
- Detected errors should be recorded and fixed prior to proceeding with development.

Reviews are generally as effective as and more cost effective than testing for finding errors and removing them. Early reviews to find errors can be greater than 10 times more cost effective than testing because errors detected early and removed are not compounded (e.g., wrong design and implementation following from incorrect specification). Reviews can also check quality factors affecting maintenance and reliability, while testing can only detect errors or prove the product works under certain preplanned conditions, not necessarily those under which it will operate. Thoroughly reviewed software products can also show a five fold reduction in maintenance costs, see [BEIZ84].

Though reviews are more cost effective for removing errors, testing is still mandatory because reviews do not remove all errors. Structured walkthroughs can remove up to 50% and Formal Inspections up to 65% of specification and design errors, see [JONES]. This still leaves up to half of the expected errors in the product.

Reviews should be an integral part of the development process and should be planned for as part of a development schedule. Reviews should be used in conjunction with testing; neither is a substitute for the other. When reviews are held at known points in a development life cycle, the review will provide product quality information (known errors and deficiencies) and quantitative progress information—product measured against where it is planned to be at that point.

5.4.1 Software Inspections

Software Inspections are a formalized, non-threatening error finding process conducted as peer reviews, where management is not present. The Inspection is led by a moderator and someone other than the author presents the material. Checklists of questions about common problem areas are used to guide the Inspection. Errors are recorded and the Inspection is not complete until the errors are resolved (outside of the Inspection meeting). Various statistics including meeting times, type of errors detected, and error rates are captured and used to improve the Inspection and development process.

The Inspections Process Standard is described in [SIPSTD] and a guidebook [SFIG] provides more information on Inspections usage. The NASA Software Assurance Standards (Draft) also sets Inspections as the review standard.

The SEPG supports Software Formal Inspections through training and by providing Inspections forms, entrance criteria and checklists as part of the Inspections class notes and in electronic format for the Macintosh.

5.4.2 Walkthroughs

A walkthrough can be any group examination of product material. It usually means the author steps or "walks" others through the product and explains what is (or is supposed to be) there. A specialized walkthrough or a "structured walkthrough" developed by Ed Yourdon is essentially a less formal Inspection. The structured walkthrough does not use checklists or keep statistics as does an Inspection.

Walkthroughs are useful for presenting overviews and material for familiarizing or synchronizing project staff simultaneously. A disadvantage of author-led reviews is that more than the actual material may get presented (design on the fly). For technical reviews of material, someone other than the author should paraphrase and present the material.

Structured walkthroughs are described in *Structured Walkthroughs*, see [YOUR83]. The philosophy and conduct for keep types of reviews is well described in the *Handbook of Walkthroughs, Inspections and Technical Reviews*, see [FREED].

5.4.3 Code Reviews

Code reviews are a form of non-executable testing. There are two basic types of code reviews: code walkthrough (which can be conducted as an Inspection) and code reading by stepwise refinement. Both types of reviews require prespecification (requirements or design specification) and the code must be cleanly compiled and successfully linked. A code walkthrough examines the code for coding errors, language errors, failure to meet standards, and incorrect implementation of the design. The latter requires that a well specified design be available.

A special type of code walkthrough, code reading by stepwise abstraction, has proven to be at least (if not more) effective than functional or structural testing at finding interface errors, see [BASILY]. In this study, the code readers were also able to more accurately estimate the percentage of problems detected. Code reading is performed by examining the application's subsystems or components code to determine their functions. By adding the component's functions the reader determines the function of the entire application. This function is then compared against the intended functionality of the requirements specification; the differences are errors. This review method requires that a good specification be available for comparison.

5.4.4 Formal Reviews

Formal reviews are project level presentations by the producer to the acquirer of interim products to determine if requirements and specifications are met, to identify problems, and to decide whether trade-offs need to be made. Phase transition reviews are specific formal reviews to decide whether to proceed to the next phase of development. Although formal reviews can be technical reviews, usually technical review results are input to the formal review process. Some attributes of formal reviews are that the review results are written as a report, they are often specifically and contractually defined, and they are usually by both the producer and the acquirer. Usually the reviewed material is baselined after review corrections are made and approved and put under CM control.

Formal reviews need not be formidable as long as the review purpose is achieved. For a small minimal risk project, the attendees can be the task requester, task manager and/or a single product developer. NASA Formal Reviews are described in a NASA *Software Formal Reviews Guidebook* (draft), see [SFRG].

5.5 Testing

5.5.1 Overview

> Testing is the operation of the software with real or simulated inputs to demonstrate that a product satisfies its requirements and, if it does not, to identify the specific differences between the expected and actual results.
> *NASA Software Assurance Guidebook.* [SAG].

Testing is the final phase of development that demonstrates product suitability and attempts to find and remove those errors not detected earlier. Although reviews are more cost effective in identifying and removing problems early in the development, they only remove half to two thirds of the resident errors. Usually they only remove about a third of resident defects, see [JONES]. Testing itself produces limited results; it does not guarantee a good product—testing only reduces "badness" by identifying errors to be removed or proving some aspect of performance that is specifically tested.

Testing can provide three measures for a product:

- That the product performs desired functions and performance
- That it doesn't perform unwanted functions

- An apparent quality indicator based on errors detected and error detection rate.

Testing can occur at four different levels over the development of the product: unit, subsystems, integration testing, and systems testing. Each level of testing has specific objectives, with different development groups having differing responsibilities (unless the development is by a single person.) Although walkthrough reviews of requirements, design, and code could be considered testing through simulation, they are addressed in section 5.4. Testing execution usually begins at the lowest level code unit and is generally done in isolation by the individual developer. As code units are assembled, integration testing continues to check increased functionality of the unit collections, subsystems, and eventually the system. Final testing is usually the Acceptance Test, a formal milestone. During the testing, all discovered errors should be recorded informally or formally depending on the test level and repaired with suitable design, review, and configuration control. After error repair, all tests at that level, and perhaps lower levels, should be run again until no errors are detected.

Test software and data should be planned and designed to be reusable and available for retesting or regression testing the application. As much as possible and practical (and appropriate for the development cost), tests should be assembled into modular components (test suites) paralleling the software tested. The suites should be capable of running under a test harness or driver so that testing consistency can be controlled. Sets of test data should also be accumulated and designed to test or prove desired capabilities.

Test and test product development is best done in parallel to product development. If an independent test team is used, they will depend on the information in product requirements and design materials for test development. An independent test team will also act as an additional assurance method, providing an important alternate viewpoint as they gather information to develop the test material. Whether done by an independent group or as part of the development group, testing development must be planned, scheduled, and designed with test products reviewed for validity and quality.

Boris Beizer has written two excellent books covering detailed high- and low-level testing approaches: *Software Testing Techniques* [BEIZ83] and *Software System Testing and Quality Assurance* [BEIZ84]. The books contain accessible and practical testing methods and approaches and are very usable for a reference or direct usage on any project.

5.5.2 Management Considerations for Testing

5.5.2.1 Factors Affecting Test Effort

The degree of testing rigorousness, amount of testing, and general approach should be determined at project startup. Numerous factors affect the test effort and approach. Among these are: project risk factors, software life expectancy, project size and staffing, development approach, and type of software developed. All of these factors are interrelated and affect the entire project and the development approach, of which testing is only one aspect.

The level of testing effort needed can be estimated similarly to other development efforts and costs. Testing development generates designs, code, test data, and documentation and should run parallel to product development. Test development products (designs and test case coverage) should also be reviewed or Inspected. Capers Jones, of Software Productivity Research, reports that, on the average, test effort consumes 15% of project costs, regardless of project size.

According to Barry [BOEHM], the primary technical risk areas are generally: developing the wrong functions; developing the wrong user interface; and developing unnecessary functionality. Determining the project's risk factors (management and technical) will shape the project approach, including the level of assurance and test efforts. The testing effort should pay special attention to the higher risk or critical areas to ensure error prevention and proper operation.

Project size, staffing, and (unfortunately) schedule often determine the level of effort which can be devoted to testing. Contractual limits may dictate that only a certain level of effort be expended on testing. The allowed level of effort may or may not be appropriate for the level needed. If a project is significantly understaffed on either development or assurance efforts, the risk rises significantly.

Software life expectancy also determines the testing effort. A one-time-use or short-lived product may not need extensive testing unless the cost of errors by any measure during that usage is high. A longer-lived product which will be enhanced and supported should have sufficient planning and effort to build a proper test framework which can be reused and easily amended for future enhancement and regression testing.

The overall guiding principle for testing (and assurance) is to prevent costly rework and errors. The higher the risk, possible cost of rework, error or failure, the more time and effort should be expended in preventing errors.

5.5.2.2 Assurance Plan and Test Plan Elements

The overall assurance <u>management</u> plan and approach should be described in the Assurance Section of the Management Plan. This will include specific activities as Work Breakdown Structure (WBS) test activities to develop, execute and evaluate test results. The WBS items should include estimates of effort and should be scheduled. Any tools developed should also be included, along with specific staff assignments and responsibilities. Any tools purchased should include activities for requirements specification and procurement lead and competitive acquisition.

Test planning approach should include as much automated support as practical, whether by purchased test tools or by built and/or reused test drivers. The Test Plan should lay out the various levels of testing from unit through acceptance testing and describe the types of testing to be used at each level. The plan should link all the tests from all the levels into an integrated approach that makes maximum use of all tests and test data sets. If possible, real world data should be used for some of the tests, especially those that may establish a known functional or performance baseline.

The specific <u>technical</u> approaches and procedures should be described in a Test Procedures document (NASA-DID-A200) [DOCSTD]. If the project is large, and a significant assurance and test effort will be supported, the specific technical procedures and activity descriptions should be described in an Assurance and Test Procedures document (NASA-DID-A000) [DOCSTD], of which the specific test descriptions can be an integral part or rolled out into the Test Procedures described above.

Adequate testing depends on an accurate and complete set of requirements and design. Testing developed from requirements validates the product against functionality that should have been developed. Testing developed from design verifies that the product logically works as intended. Testing developed from only code reduces the test effort to manual testing or automated test instrumenting after the product is developed; this inefficiency raises the cost to repair errors, many of which will be discovered only after the product is in use.

5.5.2.3 Coverage - How much is enough?

There is no "rule of thumb" about how much testing is enough—each situation depends on the development methods, testing approach, size, type, and eventual usage of the software. In an ad hoc development, testing ends when resources, schedule, or interest wanes. In a managed and engineered development, testing should be specifically targeted to the highest risk and error prone areas and its effectiveness measured by error detection rates and test coverage.

Mathematical proofs and common sense indicate that any significant program cannot have all paths and input value possibilities tested without expending an inordinate amount of time and resources. Knowing that all tests can't be performed allows testing to be oriented to the highest return.

At a minimum, the critical functional capabilities (functional testing), primary control and program flow operations (structural tests), and user interfaces of the software should be tested. Usability testing (ease of use) of features such as installation, error handling and documentation correctness, and consistency with the software should be performed. These tests are the absolute minimum necessary prior to release.

Besides selecting appropriate coverage of functional and structural testing, there are many quantitative and qualitative metrics which indicate software quality and testing effectiveness. Halstead, McCabe and Function Point metrics can provide some indicators of program complexity and therefore (from history) a prediction of existing probable errors. These software metrics and their usage are discussed in Section 3.8, Metrics.

Error detection rates, defined as errors detected per test case completed, can provide a good indication of software stability, assuming an active and effective testing phase. Discovered errors usually decline in an asymptotic curve to a consistent (acceptable) level of errors detected. The acceptable error rates are a management and engineering decision dependent on the type and use of the product. If error detection rates rise over time or remain at an unacceptably high rate, the software is not ready for release. In this case, the development method and the essential design should be examined and significantly improved.

Capers Jones suggests that for any specific type of project, given the other factors such as software type and staffing level, the number of errors which inhabit software before testing and after (residual errors) can be predicted. If adequate testing "finds" the predicted number of errors, the product may be ready for release [JONES].

5.5.3 Testing Levels

Testing is usually performed at one of three levels: unit testing is done on the lowest level of code units; integration testing is on intermediate levels of connected collections of units into subsystems and system testing is done on the completely integrated system. Each higher testing level is oriented to proving that the product's integrated parts work as a unit and to finding errors that escaped earlier tests.

5.5.3.1 Unit Tests

Unit level testing is the lowest test level, usually done by programmers on their own code or, ideally, by peers on each other's units. A unit is generally the smallest functional component of code that does something useful. A unit may be a subroutine, a function, a file, a database input screen, or some other collection of code usually produced by one programmer.

Unit testing usually occurs first, as most software problems are partitioned into numerous smaller components and developed piecewise. As those components are implemented in code, their correct implementation from design and function should be proven through testing in isolation from the rest of the system. (It is quite possible to only test all components together in a "big bang" integration; however this is not recommended.) Unit tests provide the foundation of confidence that the essential components of the subsystems and system are reasonably sound, demonstrate internally consistent behavior, and meet stand-alone expectations.

Unit testing is primarily oriented to structural testing (unit internal logic and data handling) in isolation from other units. Testing should exercise the following elements of the component:

- logic structures and paths;
- data handling including initialization, correct and incorrect input acceptance, output generation, data storage and conversions among others;

- computation including correctness and accuracy of computational algorithms.

Unit testing is usually done under the control of a software "driver" that simulates the program which would usually invoke or surround the unit. The driver should have the capability to provide and change inputs that will test the unit's internal and external behavior. The test driver and any sets of test data should be kept for reuse, as unit tests are likely to be rerun numerous times and in various combinations in support of higher level tests.

5.5.3.2 Integration Testing

Subsystems integration testing determines whether related collections of units connected (integrated) will function together and satisfy expectations. There may be multiple levels of integration and subsequent integration testing depending on the development model and design partitioning. The highest level of integration testing is the system level tests discussed in the next subsection.

Assuming that unit tests have provided confidence in the isolated units then integration testing checks the interdependencies of units working together as functional entities. The emphasis is on interface matching and unit interactions. The integration into subsystems provides opportunities for testing functional capabilities and performance requirements.

5.5.3.3 System and Acceptance Testing

System testing is the highest level of integration testing, in which all subcomponents are assembled and tested. The emphasis is on demonstrating functional capabilities and satisfaction of requirements. As with general integration testing, effective system testing depends on lower level testing to have already removed internal unit errors and interactions between integrated units (subsystems). This is the last opportunity to discover and fix errors prior to formal customer participation in testing.

Acceptance testing is usually the last phase of development and testing. Acceptance testing starts the formal turnover of the product to the customer prior to delivery. Acceptance testing and results are often a contract deliverable, usually with the customer in attendance or in review of the test results.

5.5.4 Testing Approaches

Testing approaches for hierarchical systems will generally mirror the development's implementation approach. The most common approaches are top-down vs. bottom-up and phased vs. incremental. Each approach has advantages and disadvantages and seldom does any development use only one approach.

A top-down approach develops and tests the highest and usually most visible components first and proceeds to build and test components at progressively lower levels. Top-down development requires "stubs" to simulate the components not yet developed. A bottom-up approach develops the lower levels first and combines them until a complete system is assembled and tested. Bottom-up development requires software "drivers" to simulate the as yet undeveloped higher control structures. Yourdon and Constantine favor a top-down approach, primarily because users, managers, and programmers are pleased to see early preliminary (but not complete) versions of the product. An argument against a purely top-down approach for the case in which critical risky sections are at the bottom (or last) in the development schedule; in this case these portions should be demonstrated as early as possible.

A phased approach develops and assembles a group of components into a larger functional whole. An incremental approach is similar, except that additions are added one at a time. The incremental approach allows errors to be more easily isolated and fixed.

5.5.5 Testing Methods

Regardless of the testing level or approach, specific test methods must be applied to detect errors or demonstrate functionality in software. The following methods are generally applicable to most software levels and testing approaches, although some of the methods are more applicable to specific levels or approaches (e.g., pilot case testing is applicable primarily to system level testing).

5.5.5.1 Functional Testing (Black Box)

Functional testing exercises the interfaces and the external responses of the component (or system) to ensure requirements or expectations are met. Functional testing treats the component as though no knowledge of the internal logic or structure is available, and relies on the functional description of what it is supposed to do. System level and acceptance testing primarily use this testing method, although it is applicable to all testing levels and approaches.

Adequate coverage for black box testing includes selection of test inputs or stimulus for all functional capabilities and should address a range of behavior such as:

- normal use of the function
- abnormal but reasonable use of the function
- abnormal and unreasonable use of the function

If possible, the user should be involved in generating the expected answers and results from the test cases. The use of real-world data and comparison of validated results from an existing application provides a productivity improvement opportunity.

The production of a preliminary User Guide early in a development can be a form of functional testing that can help discover user concepts and expectations of the system's appearance and capabilities. The user guide only deals with the interfaces: inputs, outputs, and visible response of the system.

5.5.5.2 Structural (Glass Box) Testing

Structural testing exercises the internal logic, paths, behavior and algorithms to ensure correct implementation of the design. Structural testing is based on a design knowledge of internals of the system or component. Test inputs and stimuli are selected specifically to exercise and find problems in the logic paths, intermodule interfaces, and shared and passed data structures.

Structured Testing is a specific methodology that was developed by T. J. McCabe (of McCabe Complexity metrics) in the mid-1980s. Complexity analysis module ratings and the charts of module structure, branches and logical paths are used to generate tests to exercise the main logic paths in a program. Part of the testing is "instrumenting" the software with a testing tool to record which paths have been traversed during the test cases. Test cases to exercise untraversed paths are added to the test plan and the tests are rerun until coverage is satisfactory.

A drawback with structural testing is that it is usually impossible or not practical to test all possible inputs and paths for a real world problem. Additionally, glass box testing cannot define needed elements that are missing, as it only tests what does exist.

5.5.5.3 Proof of Correctness Testing

Correctness proofs are mathematically-based deductions, formed from logic theory. The system or product requirements are stated in a formal mathematical language. Then, for possible inputs the requirements statements are logically examined to see if correct outputs will be produced. According to Beizer [BEIZ84], the limitations are that there is no assurance that the specification is correct, that the proofs are expensive, and that there is, generally, limited applicability.

Commercial equation checkers, such as MACSYMA and Mathmatica, which will verify an equation in symbolic language and generate FORTRAN, C, or other code to implement the equation, are available.

5.5.5.4 Concurrency Testing

Concurrency testing is used to check a system's ability to support simultaneous activities against shared or common resources. For concurrency testing in database technology, record and data locking are tested to ensure data integrity for multiple simultaneous access. Two conditions to test are: avoidance of deadly embrace (deadlock—where multiple processes are "stuck" awaiting each other's response) and proper operation of queuing and wait states response. Stress testing support and multiple access simulation may be necessary to provide a sufficiently overloaded environment where possible contention of resources may occur.

5.5.5.5 Stress and Performance Testing

Stress testing is used to help detect problems associated with load, performance, and resource constraints. Stress testing is not correctness testing, but oriented to discovering at what level the product breaks. Capabilities required for stress testing are simulated or actual system overloading beyond normal operational limits which can cause contention for system resources. Stress testing usually simulates the peak loading expected and then continues to add loading or remove resources until the system seriously degrades or breaks. Automated test tools can help greatly in simulating user and other types of loading.

Performance testing is done to 1) demonstrate that system performance meets specifications, 2) tune the system, 3) determine performance limiting factors, and/or 4) project future system load-carrying capability, [BEIZ84]. Performance testing requires a relatively error-free system so that the performance problems can be isolated from functionality and implementation problems. Effective performance testing requires that performance objectives be well specified (and realistic), the system (hardware and software) be operational, all system parameters and loads be controlled, and variables should be changed (and changes measured) one at a time.

For system tuning, performance limiting factors should be eliminated or modified to achieve a required performance within given constraints. Stress and Performance testing with system tuning can be done repeatedly, back to back, until optimal conditions are achieved.

5.5.5.6 Usability Testing

Usability testing is performed to evaluate ease of use. It is specifically user oriented testing, usually by a person unfamiliar with the product. Pilot case testing, described below, is a mechanism to evaluate usability. Specific areas to address include: ease of installation, documentation, consistency, misunderstandings, changes required to other programs and/or systems; and access to support. If possible, real users in actual environments provide the best usability testing environment.

Under the current productivity, quality, and TQM initiatives, usability testing and customer satisfaction is assuming even more importance. Involving the user as early as possible through rapid prototyping for discovering the real requirements, paying attention to human factors, and listening to the more subjective user preferences should pay off in higher customer satisfaction.

5.5.5.7 Regression Testing

Regression testing is used to ensure that any change to the product has not introduced "side effects" or unanticipated changes. Regression testing runs previously proven tests, test software, and test data against existing product capabilities to generate test results that can be compared to previous results. Use of automatic test runners with results comparison capability and file comparator products make the testing effort more productive and accurate.

Regression tests are essentially retests which require discipline in the testing process so tests and groups of tests can be exactly repeated. Regression tests work best when the tests are automated and should be repeatable with similar results expected. When dissimilar or unexpected results are obtained from the same test sequence and the tests have not changed, something other than the tests have changed. Regression tests, then, provide a benchmark or standard on which product performance or capability is compared.

5.5.5.8 Pilot Case Testing

Pilot case testing is used to evaluate products from a user point of view. Pilot case testing is usually referred to as Alpha testing or Beta testing. Alpha testing is product testing using in-house staff acting as users in the environment where the product will be used. Testing should include product installation and initialization and, if possible, be accompanied by any documentation. The testers should be familiar with the actual user requirements and environments. The testing will evaluate the product's functional capabilities in as-near a real user environment as possible.

In beta testing, the product is released to a limited and controlled group of "friendly users" for use in actual operational environments by real users. Users are aware that the product is still in the test phase. The beta test product should be as complete as possible and include documentation.

Both alpha and especially beta testing should be well supported by a robust problem reporting system, a consistent and reliable CM and release generation system, and rapid response and fix for discovered errors. Even though the beta testers are "friendly users," this is their first real exposure of the almost completed product. Poor first impressions require enormous amounts of work to overcome, and if the product is not really ready for beta test, may be counterproductive.

5.5.5.9 Object-Oriented Testing

There are few guidelines for object-oriented testing at the time of this writing. While testing of object-oriented components can be viewed and tested similarly to procedural-code components, the use of classes and inheritance can add complexity which must be addressed for adequate testing.

Testing object-oriented systems at the lowest routine levels and at the highest system level is straightforward and similar to traditional unit and system testing methods. Low-end routines are tested to ensure that data responses and logic responses are as expected. Highend system testing, using "black box" or requirements specifications based functional testing, ensures that visible performance, data, and functional requirements are satisfied.

Testing of object-oriented programs presents several problems not encountered in sequentially executed programs. Traditional testing provides input to a process and checks the output against known and expected results. Lower level process units can be integrated into more complex units, all of which have dependable sequences of execution. Object-oriented program and component testing can be more difficult since objects have attributes of both data and methods, and, with inheritance, objects can be combined and executed in an arbitrary sequence.

The following material is extracted from research into the academic software engineering literature currently available on this topic and is included with the hope that it may be of some help to those wrestling with the new problems unique to the testing of object-oriented software.

Harrold and McGregor [HARROLD] and Smith and Robson [SMITH] use a four-level model for developing a testing strategy for object-oriented work. This model uses the algorithm, class, cluster of classes, and system for levels of abstraction. Algorithmic is the lowest level routine which manipulates data. Class is the interactions of routines and data encapsulated in a class. Clusters are the interactions of cooperating classes. System level is all classes and programs necessary to run the system.

Algorithmic and system level testing is similar to procedural code testing. Class testing is more difficult because the services and data structures of the object contain the state and no ordering of the execution is predictable. Traditional sequential data input–process–output testing models aren't adaptable. "Functional testing techniques will not work since there is no *test set* to 'run' the code with. Structural techniques are also not directly applicable to a class since it is difficult to analyze control flow or data flow through it." [SMITH]

[HARROLD] takes the approach that if a class can be validated, than inheritances from that class can reuse the testing information from the parent. With this information, only the new or replaced attributes and any inherited attributes affected in the new context need to be tested, not the entire subclass. This does require that a comprehensive testing of the parent class be performed and a history kept.

Both [HARROLD] and[SMITH] have focused their research on intra-class testing and have left interclass or class cluster testing for further research.

5.5.6 Recommended Test Tools

Two specific support tools that every project should acquire or produce are a test driver to consistently run test cases and an automated release build capability. The test driver should support any level of testing and be table or file data driven. The test data for any specific test case should be saved and available for regression testing. An automated build tool (such as UNIX 'make') is part of a controlled release generation capability which should be used to consistently build release versions from known source and object libraries. Controlling changes, tracking module's versions, and building from known source is especially useful during integration and system testing to isolate changes and the side effects of those changes which introduce anomalous behavior.

5.6 Problem Reporting

Each task and project which develops and/or supports a product or provides a service should have a procedure in place for reporting and recording problems, errors, and change requests. The procedure may be automated and mechanized or it may be a manual paper system. The most important point is that detected problems (especially those detected by the customer) be resolved—and, for customer detected problems, that the customer be notified of the resolution.

The extent of a problem reporting system will depend on the product being supported, the development approach, and the management approach. Small projects may only need an informal system to support problem reporting and ensure the problems are resolved. Larger or higher risk projects may need a formalized reporting and tracking system which is Configuration Control managed.

The minimum information which should accompany a reported problem are: the problem date, description, urgency, and who reported it, so that the problem can be duplicated or more information can be acquired. Much more information can be collected or added later, but other data collection will depend on the needs of the development or management approach. Data that may be added by the analyst or maintainer and are useful for managing the development process are as follows:

- Problem identification - a unique identifier for tracking and history.
- Severity - a measure of the level of the problem.
- Description of problem - what is the actual problem; this may not be the same as the reported problem.
- Resolution - what actually was done to resolve the problem. This may include who was assigned, CM information.
- Problem location - which software units contained the problem(s).
- Reason problem occurred - what was the cause, e.g., bad requirements, coding error, improperly applied fix, documentation error.
- Problem report date, assign date, and resolution date.
- Time spent to analyze and repair.

Well managed projects can use problem reporting metrics as indicators of software health and development progress, or as a means to improve the development process and methodology. (See Section 3.8 , Metrics). This information provides greater visibility into the development process and software, and thus allows more informed decisions about the direction of the development and problem areas in the software.

Section 6

CONFIGURATION MANAGEMENT

Designing configuration management procedures is an exercise in compromise. You must walk the thin line between chaos and stifling bureaucracy. The procedures must be tight enough that most of the energy of the programming staff is devoted to productive work, and paradoxically must also be loose enough that most of the energy of the programming staff is devoted to productive work. ...A good rule of thumb is not to try to control every conceivable situation ...allow escapes for manual intervention... Remember at all times—the goal is to provide a stable programming environment so that work can get done. Configuration management is a means, not an end.

Wayne Babich, *Software Configuration Management* [BABICH]

6.1 Overview

Configuration Management (CM) is the process of identifying specific work items, defining known versions of work items (baseline), controlling changes to the baseline and keeping records of the version and changes to it. CM is applicable to all parts and phases of a development process, from the documented requirements and design through software testing, acceptance and continuing operation and maintenance.

Basic terms used in CM [SWASTD] are as follows:

- baseline - an established version of a work product used for controlling future changes, subjected to configuration control.
- configuration control - a systematic control process for a work product and changes to it after a formal establishment of a baseline.
- configuration item - an aggregation of hardware or software which is designated for configuration management.
- configuration management - the total process of baselining work products, controlling changes to configuration items and their baseline work products and maintaining accounting records of the change.

The CM activities should be appropriate for the items controlled. The underlying purpose of CM is to keep track of what is being developed, tested, changed or released. The CM process should be simple and practical for small projects or can be quite complex and require many checks and controls and require several full time staff for complex or high risk projects.

How much CM is enough? Some questions that can help guide the amount of CM effort are:

- Who are the users, how many are there and what do they need?
- Do the developers waste time trying to figure out "What program is this?"
- How long will the software or product be in use?

- What risk factors apply to the software or the controlled configuration?
- How important is it that only controlled or approved additions to the software be made?

CM is an integral part of a coherent development process; it is not added on as an imposition to development. CM, if done properly, aids and speeds development and testing by providing an orderly and known environment and by reducing confusion and duplication. Knowing the changes to baseline configurations aid testing and regression testing by providing differences between releases and versions.

6.2 Configuration Management Process

6.2.1 Configuration Identification

Configuration identification is the first and essential action in CM. It is basically selecting and identifying by name the items to be controlled. Controlled items are usually functionally grouped—for example, a group of software units that work together such as a disk controller, an application program, or a system subcomponent. Both the group and its individual units are identified by name and composition. Some types of software products that can be controlled are source, object, executable and component libraries, testing software, utility software (such as scripts, 'make' files, etc.), databases, schemas, data files, documents, problem reports and change requests.

After the "what" to control has been identified, the "when" to control needs to be defined. Selecting items to be controlled and at what point to control them defines a specific baseline. Usually, version numbers uniquely identify groups and constituent components of the controlled items. Formal reviews, usually at the end of a phase, provide the "stamp of approval" that make the baseline official. Formal configuration control has defined major baselines throughout the development cycle with specific names: completed, approved requirements become the Allocated Baseline, designs and test material become the Development Baseline, and tested and released software and its documentation become the Product Baseline. For most projects, baselines can be established at the end of or within any defined phase, such as, requirements, design, implementation, testing and/or acceptance.

For rapid prototyping and other iterative methods, there will be numerous informal baselines. These informal baselines are established and controlled within the development group to keep known versions distinct. At the end of each iteration the reviewed material should be saved off and either baselined and/or archived with a list of changes and/or problems found.

Baselining, simply put, is reserving or setting aside for control purposes, a given set of items at a specific point in time. Each of the items should be uniquely identified by version number or some other means. Usually, a preliminary baseline is established prior to a review on materials. After review changes are made, a final baseline of the material is done and development proceeds from *that known composition configuration*. A copy of the baseline material is usually archived or set aside for future reference.

6.2.2 Configuration Control

Configuration control describes responsibilities for controlling defined configuration items during their lifetimes, what paperwork and forms are required to document configurations and changes, and who is authorized to allow changes to controlled items. The actual flow of change requests, evaluation, approval, accomplishment and baseline updates are described in the next section.

For any project or activity, responsibility for control of configuration items should reside with a single individual, even if a group performs the process.

Products should be known and under control at all times. The configuration control can be formal or informal. Formal control applies to products delivered to or approved by the customer. Unless assigned to the developer, control belongs to the customer. Changes to these items require approval by the customer.

Between baselines and within the development environment, informal control rests with the developer. The approved development plan provides the authority and directs the enhancement or development activities leading to the next formal review and delivery of the product to the customer.

When change requests are received and evaluated, they may be assigned a category or class dependent upon the scope of the change and whether formal or internal control is applicable. The following descriptions are pertinent:

> Class 1 changes may change program requirements or negotiated costs, or alter schedules and require customer approval.
>
> Class 2 changes clarify or correct errors to approved products, but aren't Class 1. These require customer concurrence to accomplish.
>
> Class 3 changes affect software requirements, user interfaces, architecture or data structures under development or between baselines (but are not class 1 or 2). Class 4 corrects or clarifies errors.

Change requests (CR) provide a format for requesting a change, providing justification and evaluation of change extent, approving or authorizing and accomplishment sign-off when complete. The CR information is used by the change authority to decide whether to approve the change or not.

Change authority and approval is done only by the organization that holds the configuration control. In larger development environments, the change authority is a Change Control Board (CCB) which may be composed of both the developer and the customer which jointly make change control decisions and approvals. The CCB makes decisions based on a Change Request, any associated information in the context of the project plan (budget and schedule) and objectives.

6.2.3 Change Control Flow

Change Control flow and processing describe the responsibility path that a change follows from its initiation through actual update of the baseline. The essential elements of the flow will be the same regardless of the size of the CM effort and number of participants. CM, QA, development staff and the customer may all participate in change control. In small projects this may only be one person for all parts of the process. The degree of formality, tracking and control are also dependent on the project size.

Figure 6.3-1 provides a depiction of a *possible change flow process* in a project with separate CM, QA, development groups and a combined customer-developer Change Control Board (CCB). The activities can be divided into three major activities:

- investigation
- approval and
- implementation.

CM performs primarily a control and tracking function. Any project's CM may be more or less complex, depending on project requirements.

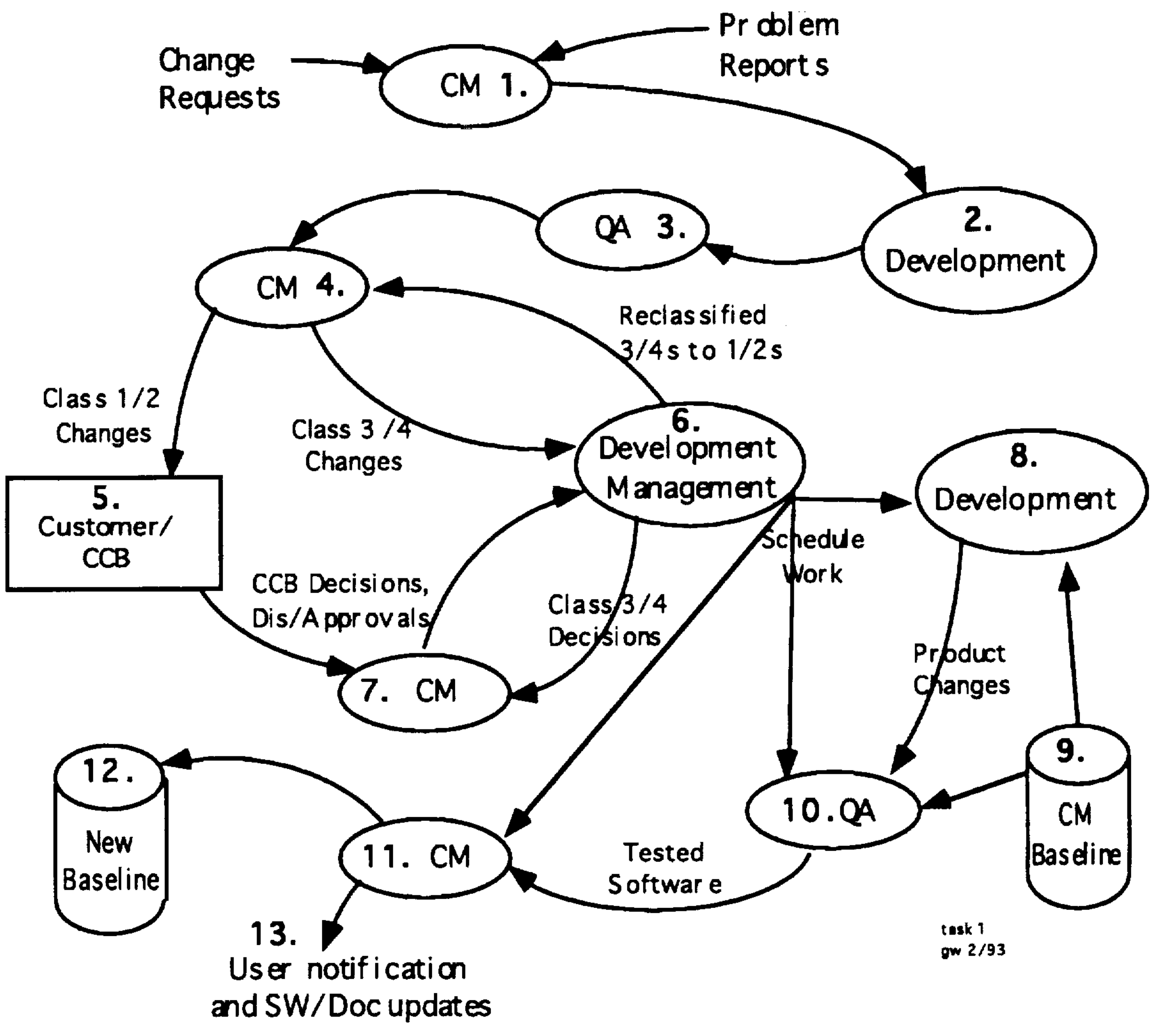


Figure 6.3-1 Possible CM Change Flow

In the figure, Software Problem Reports (SPRs) and Change Requests (CRs) (in this discussion we will refer to both as "requests") are received form various sources and (1) logged by CM. The requests are forwarded to development for (2) problem evaluation and work estimate. QA receives the request for evaluation and (3) adds its evaluation and work estimate for testing and validation. CM (4) adds its configuration estimate to possibly generate the applicable baseline and to re-establish a new baseline if necessary and release the changes. CM also logs the request's progress. Depending on the classification of the request, CM routes the request to the CCB (5), if customer approval or concurrence needed, or (6) to development management for scheduling and incorporation into regular work. Class 3 and 4 requests are also (7) sent to CM for logging for progress tracking or closure for disapproved items. Class 3 and 4 items may be reclassified higher and re-input to the CCB through CM (4).

CCB (5) evaluates requests against project objectives and resources and approves for development or disapproves. The decisions are routed to development management through (7) CM for logging and, if no action is required, closing the request. Management schedules the work for (8) development, (10) QA and (11) CM as part of a scheduled release or as an immediate fix.

After the work has been scheduled, development (8) performs the work and unit/integration tests and passes the changes to (10) QA for any independent and system testing. As necessary, SPRs or CRs may be generated (back to (1)) if a revealed problem is outside the scope of the approved change. Completed and tested software along with any documentation is forwarded to (11) CM for logging, closing the request and establishing (12) a new baseline. CM also provides (13) change notification, release of software changes and new/updated documentation to the user community.

6.3 Suggested CM Procedures by Project Size

The following sections describe possible CM activities for three different sizes of software projects at Ames. The assumption is made that no high risk software is being produced so a moderate level of QA and CM support is needed. The three scenarios present minimum recommended CM activities; more effort can be spent on CM. There are several standard procedures which should be common to any size activity or task to provide minimum software control and security to data and work. These are:

- Separate the development, test and release versions of the product. Make sure each is defined and well known. If possible separate the activities by assigning responsibility or at least by doing them at different times.
- Make the compile and build process (or any regularly performed activity) automated and consistent.
- Back up and archive at a remote site the released version (source, utilities and executable); uniquely identify it and make sure related documentation and design information are available.
- Back up the work in progress regularly, perhaps incrementally each day.

6.3.1 Small Project CM

Minimal CM activities for any small project:

- Requirements (researcher requests) and discovered problems should be written and saved in a readily available notebook. Major problems, changes to functionality and schedule changes should be discussed with the customer.
- A source control system and an automated 'make' should be used to keep version differences and provide a consistent build and link cycle.
- Use separate source directories to generate object libraries to speed up build time and provide a basis for reuse.
- Reported problems and change requests should be logged when received, tracked to completion and closed when done.
- A standard problem report and change request form with specific questions should be used to help pinpoint user problems and speed the solution.
- Any release should go through a standard integration, test and release process separate from the development activity. For low risk tasks, this might be accomplished by using a separate test and release directory.

6.3.2 Medium Project CM

Minimal CM activities for any medium size project:

- Problem reports and change requests should be logged and kept open until accomplished or closed. The log should help drive the Project Manager's work list and schedule.
- An informal but planned release cycle should be used with planned dates and known enhancements.
- Use a code or version control system (e.g. SCCS)
- Hold development technical reviews/inspections to establish requirements baseline.

- Release should include version list, design or structure notes, machine environment requirements and build instructions with any associated utilities, 'make', build or script files.
- Archives of the last 3 releases with the documents, source code and executables necessary to run it.

6.3.3 Large Project CM

Minimal CM activities for any large size project:

- One team member should be assigned as CM librarian/build custodian.
- Define a formal release cycle with publicized release, enhancement and change notification
- Use a code or version control system (e.g. SCCS).
- Use a problem reporting and tracking system.
- Use a formal technical reviews/Inspections establish baselines in development products.
- Follow formal CM change control and approval of changes.
- Keep a list of installed customers/sites for change/enhancement/release notification.
- Archive of all supported releases should be readily available and include the documents, source code, utilities, libraries, test data and executables necessary to build, test and run. Archives of older versions should be stored and retrievable.

6.4 CM for Rapid Prototyping Projects

CM for rapid prototyping and other iterative developments is similar to sequential developments for the major deliverable milestones. The developer has internal CM responsibility for the products of each iteration (between the major milestones), just as with a sequential development. Problem lists and results of the iteration reviews and internal technical reviews or Inspections must be kept, maintained and resolved

When the prototype iterations are complete and approved by the customer, the prototype is formally baselined and controlled. The design is then completed, reviewed and corrected, delivered and formally controlled just as in a sequential development. The prototype model is then fully implemented to the design, tuned, optimized, tested and delivered for Acceptance Testing and CM baseline and control.

Section 7

ABBREVIATIONS AND ACRONYMS

This section contains an alphabetized list of definitions for special abbreviations and acronyms used in this volume.

AHB	Ames Handbook
AI	Artificial Intelligence
CASE	Computer-Aided Systems Engineering
CCB	Configuration Control Board
CCF	Central Computing Facility
COTR	Contracting Officer's Technical Representative
CM	Configuration Management
CTO	Contract Task Order
DFD	Dataflow Diagram
DID	Data Item Description
DoD	Department of Defense
ERD	Entity-Relationship Diagram
GFE	Government-furnished equipment
LOC	Lines of code
NHB	NASA Handbook
OCM	Object Control Matrix
OO	Object-oriented
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
QA	Quality Assurance
RP	Rapid Prototyping
RTSA	Real-Time Structured Analysis
SA	Structured Analysis
SEPG	Software Engineering Process Group (part of Task 1)
SMAP	(NASA) Software Management Assurance Program
SPR	Software Productivity Research, Inc.
STD	State-Transition Diagram
TBD	To be determined (at a later date)
TQM	Total Quality Assurance
V&V	Verification and Validation
WBS	Work Breakdown Structure

Section 8

GLOSSARY

This section contains an alphabetized list of definitions for special terms used in this volume.

Assurance - includes any and all activities, independent of organization conducting the activity, that demonstrate the conformance of a product to a prespecified criteria (such as to a design or to a standard).

Baseline - an established version of a work product used for controlling future changes subjected to configuration control.

Configuration control - a systematic control process for a work product and changes to it after a formal establishment of a baseline.

Configuration item - an aggregation of hardware or software which is designated for configuration management.

Configuration management - the total process of baselining work products, controlling changes to configuration items and their baseline work products and maintaining accounting records of the change.

Code Q - NASA Office of Safety, Reliability, Maintainability, and Quality Assurance.

Critical Design Review - the phase transition review for the Detailed Design life-cycle phase.

Data Item Description - the table of contents and associated content description of a document or volume.

Deliverable - a contractually defined or normally expected product.

Life-cycle - the period of time that begins when a product is conceived and ends when the product is no longer in use.

Phase - a defined process step with required inputs, defined activities and specified outputs.

Quality assurance - A subset of the total assurance activities generally focused on conformance to standards and plans. In general, these assurance activities are conducted by the SRM & QA organization.

Risk - the combined effect of the likelihood of an unfavorable occurrence and the potential impact of that occurrence.

Risk Management - the process of assessing potential risks and reducing those risks within dollar, schedule, and other constraints.

Roll-out - A mechanism for recording sections of a document in physically separate volumes while maintaining traceability and links. When using roll-out, a volume is subordinate to a parent document or volume.

Testing - the process of exercising or evaluating an information system or component by manual or automated means to demonstrate that it meets specified requirements or to identify differences between expected and actual results.

Validation - 1) assurance activities conducted to determine that the requirements for a product are correct; i.e. to build the right product. 2) (IEEE Std 729-1983) the process of evaluating software at the end of the software development process to ensure compliance with software requirements.

Verification - 1) assurance activities conducted to determine that a product is being built correctly in accordance with design and requirements specifications; i.e., to build the product right. 2) (IEEE Std 729-1983) "The process of determining whether or not the products of a given phase of ... development ... fulfill the requirements established during the previous phase."

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1993	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE Software Engineering Guidebook	5. FUNDING NUMBERS NAS2-13210
6. AUTHOR(S) John Connell and Greg Wenneson	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Sterling Software, Inc. 1121 San Antonio Road Palo Alto, CA 94303	8. PERFORMING ORGANIZATION REPORT NUMBER A-93135
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA CR-177625

11. SUPPLEMENTARY NOTES
Point of Contact: Robert Carlson, Ames Research Center, MS 233-10, Moffett Field, CA 94035-1000
(415) 604-6036

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category – 61	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Software Engineering Guidebook describes SEPG (Software Engineering Process Group) supported processes and techniques for engineering quality software in NASA environments. Three process models are supported: structured, object-oriented, and evolutionary rapid-prototyping. The guidebook covers software life-cycles, engineering, assurance and configuration management. The guidebook is written for managers and engineers who manage, develop, enhance and/or maintain software under the Computer Software Services Contract.

14. SUBJECT TERMS Software-engineering, Software-methodology, SEPG	15. NUMBER OF PAGES 68
	16. PRICE CODE A04

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT